Studiengang Informationstechnik

Bachelorarbeit

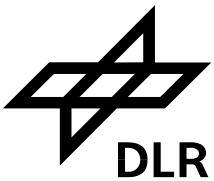
Bearbeitung: 22. Juni 2015 - 14. September 2015

Thema: Entwicklung einer grafischen Modellierungssprache für ein ereignisgesteuertes Echtzeit-Laufzeitsystem

von **Tobias Franz**

- Matrikelnr.: 9148081 -

- Kurs: TINF12ITIN -



Deutsches Zentrum für
Luft- und Raumfahrt e.V.
in der Helmholtz-Gemeinschaft

Standort: Braunschweig

Simulations- und Softwaretechnik
Abteilung: Software für Raumfahrtsysteme und interaktive Visualisierung

Betreuer: Benjamin Weps

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich
die vorliegende Arbeit selbstständig und nur unter
Verwendung der angegebenen Quellen und
Hilfsmittel angefertigt habe.

Braunschweig, der 14. September 2015

Zusammenfassung

Um große Datenmengen unter Echtzeitbedingungen auswerten zu können, müssen unterschiedlichste Softwarekomponenten und Algorithmen verbunden werden. Mit dem Tasking Framework wird die Kommunikation und ereignisgesteuerte Ausführung solcher Komponenten vereinheitlicht. Da die Konfiguration dieses Frameworks sehr aufwendig ist, wurde eine grafische Modellierungssprache entwickelt werden, um den notwendigen Code generieren zu können.

Diese Arbeit beschreibt die Evaluierung vorhandener, grafischer Modellierungssprachen und schließlich die Erstellung einer, auf das Anwendungsgebiet der ereignisgesteuerten Kommunikation, spezialisierten Sprache. Mehrere Diagramme ermöglichen dabei die Beschreibung der Software unter verschiedenen Gesichtspunkten.

Zur Umsetzung eines Editors wurden mehrere Werkzeuge zur Erstellung von Diagrammeditoren verglichen und schließlich eine Kombination aus dem Graphical Modeling Framework und dem darauf aufbauenden EuGENia verwendet. Der so erstellte Editor lässt nur wohlgeformte Diagrammanordnungen zu und ermöglicht eine Validierung der Element-Parameter.

Um die modellgetriebene Entwicklung an projektspezifische Anforderungen anzupassen, unterstützen Sprache, Editor und Code-Generator die Erstellung und Verwendung neuer Elemente über ontologische Konzepte.

Abstract

To compute a large amount of data in real-time systems several software components and algorithms need to be connected. The Tasking Framework enables communication between components and the event-driven execution of such components. Due to the complexity of the configuration of this framework, a graphical modelling language was developed to generate the required source code.

This thesis describes the evaluation of existing graphical modeling languages and, eventually, the definition of a language specialised in the scope of event-driven communication. Multiple diagrams facilitate the description of the software from various perspectives.

To implement a diagram editor, several frameworks were compared and, finally a combination of the graphical modelling framework and EuGENia was selected. The editor allows only well-formed combinations of diagram elements and provides a validation of the element parameters.

To adapt the model-driven development to the project-specific requirements, the language, editor and code generator support the dynamic creation of new elements using ontological concepts.

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

AADL	Architecture Analysis and Design Language
ATON	Autonomous Terrain-based Optical Navigation
DLR	Deutsches Zentrum für Luft- und Raumfahrt
DSL	Domain-Specific Language
EMF	Eclipse Modeling Framework
FIFO	First-In-First-Out
GMF	Graphical Modeling Framework
INCOSE	International Council on Systems Engineering
MDA	Model-driven Architecture
MDSD	Model-Driven Software Development
MOF	Meta Object Facility
OMG	Object Management Group
PIM	Platform-independent model
PSM	Platform-specific model
SysML	Systems Modeling Language
UML	Unified Modeling Language
W3C	World Wide Web Consortium
XMI	XML Metadata Interchange
XML	Extensible Markup Language

1 Projektumfeld und Motivation

Diese Arbeit ist im Deutschen Zentrum für Luft- und Raumfahrt in Braunschweig in der Einrichtung Simulations- und Softwaretechnik entstanden [DLRc].

1.1 Software für Raumfahrtsysteme und interaktive Visualisierung

Die Abteilung Software für Raumfahrtsysteme und interaktive Visualisierung hat zwei große Aufgaben, erstens die wissenschaftliche Visualisierung, zweitens die Erstellung von Software für Raumfahrt. Die Visualisierung beschäftigt sich unter anderem mit der Simulation von aerodynamischen Strömungs- oder Raumfahrtdaten, bei der z.B. Satellitenkonfigurationen oder Planeteneigenschaften dargestellt werden. Diese Visualisierungen sollen unter anderem genutzt werden, um Astronauten zu schulen. Der zweite Aufgabenbereich ist die Erstellung von Software für die Raumfahrt. In diesem Teil wird sowohl Software zur Unterstützung der Erstellung von Raumfahrtkomponenten erstellt, als auch Software für die Onboard-Systeme von Satelliten oder Raumsonden.

1.2 Ereignisgesteuerte Echtzeit-Laufzeit-Umgebung im Raumfahrtbereich

Aktuelle Raumfahrtmissionen haben in der Regel eine vorbestimmte Flugbahn, die mit viel Rechenleistung auf der Erde berechnet wurde. Um eine größere Flexibilität zu ermöglichen und um in unbekannten Regionen Flugbahnen für z.B. Landungen zu bestimmen, müssen diese jedoch live berechnet werden. Aus diesem Grund müssen zukünftige Raumfahrtmissionen große Datenmengen noch während dem Flug durch verschiedenen Komponenten und Algorithmen auswerten können. Dabei werden Daten zur Steuerung zuverlässig innerhalb bestimmter Zeiträume benötigt, es herrschen Echtzeitanforderungen. Um Trajektorien in Echtzeit zu berechnen, werden viele Sensoren benötigt, deren Daten von verschiedenen Algorithmen ausgewertet werden. Die Onboard-Software eines solchen Raumschiffs muss dabei die

Zusammenarbeit und Datenübertragung der Komponenten gewährleisten und entscheiden, wann welche ausgeführt werden.

1.2.1 Echtzeitgesteuerte Prozessausführung im Tasking Framework

Eine Möglichkeit zur Zusammenführung einer solchen Komponentensoftware ist das Tasking Framework [MLG]. Das Tasking Framework bietet eine Schnittstelle für die Aufgaben (engl. Tasks) die parallel ausgeführt werden müssen und verwaltet die Datenübertragung zwischen diesen Tasks. Die Tasks können dabei ausgeführt werden, wenn eine bestimmte Kombination an eingehenden Daten zur Verfügung steht oder ein Event ausgelöst wurde.

Das Tasking Framework stellt fünf abstrakte Klassen bereit, die der Entwickler implementieren muss. Will der Entwickler einen Algorithmus als Task umsetzten, muss er die Task-Klasse überschrieben und eine „$execute()$“-Methode implementieren. Über die Implementierung einer TaskChannel-Klasse wird bestimmt, wie die Daten zwischen den Tasks übertragen werden. Das Tasking Framework verwaltet dabei nur die Ereignisübergabe. Ein TaskChannel kann Daten für mehrere Tasks gleichzeitig bereitstellen, diese können dabei verschiedene Eingangsparameter haben. Ein Task kann zum Beispiel erst ausgeführt werden, wenn eine bestimmte Anzahl an Daten in den TaskChannel geschrieben wurden. Um unterschiedliche Eingangsparameter für verschiedene Tasks mit dem gleichen TaskChannel verwenden zu können, gibt es die Klasse TaskInput. Neben der Ereignisübergabe beim Eintreffen von Daten können TaskEvents auch periodisch Signale an Tasks senden.

Zur Verdeutlichung ist in Abbildung 1 eine mögliche Konfiguration des Tasking Frameworks dargestellt. Die in blau dargestellten Softwarekomponenten sind als Task implementiert, um sie parallel ausführen zu können. Die in grün dargestellte Kamera ist ein Sensor, der Bilder für eine Kraternavigation und einen Featuretracker bereitstellt. Diese Module werden von der Kamera getriggert, jedoch läuft der Featuretracker doppelt so oft wie die Kraternavigation, da diese immer erst bei jedem 30ten Bild ausgeführt wird. Die TaskInputs werden hier benötigt, da die beiden Tasks verschiedene Eingangsparameter für denselben TaskChannel haben. Die Kraternavigation benötigt zusätzlich noch die letzte Position des Navigationsfilters um einen Startpunkt für den Abgleich der Krater zu haben. Die Ergebnisse der beiden ersten Tasks werden in dem Navigationsfilter zusammengerechnet. Dieser Task benötigt dabei keine bestimmten eingehenden Daten, sondern er wird über das TaskEvent als Timer periodisch

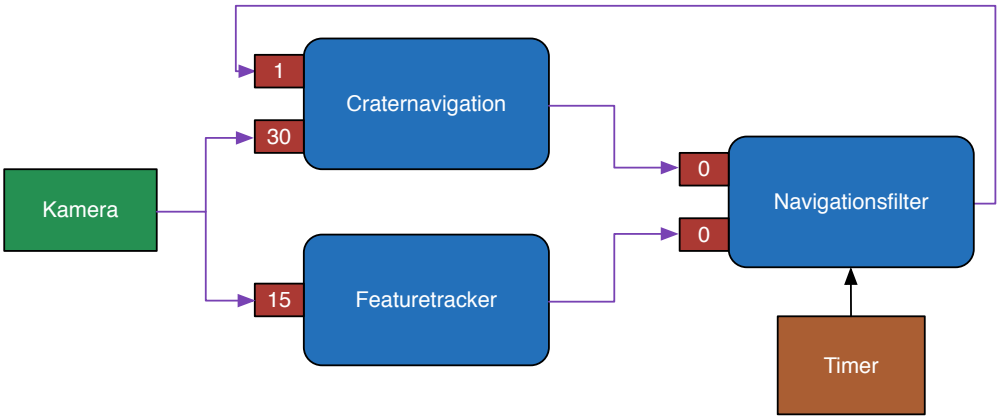


Abbildung 1: Darstellung einer Beispielkonfiguration des Tasking Frameworks. Grün sind Sensoren, blau Tasks, rot TaskInputs und orange TaskEvents dargestellt. Die Verbindungslinien entsprechen dem Datenfluss und stellen die TaskChannels da.

ausgeführt.

In realen Projekten werden in der Regel deutlich mehr Sensoren und Auswertungskomponenten benötigt. Dies macht die Softwarestruktur deutlich komplexer, da so neben neben den Tasks auch Schnittstellen und Datentypen für die einzelnen Module implementiert werden müssen.

Da das Onboard-System ein eingebettetes System ist und es im Raumfahrt-Kontext eingesetzt wird, wird eine Code-Richtlinie für kritische Systeme verwendet [con08]. Aus diesem Grund kann zum Beispiel keine dynamische Speicherverwaltung verwendet werden. Um die Übertragung verschiedener Datentypen über die TaskChannels zu ermöglichen, werden diese in der Regel als Templateklassen umgesetzt. Das Zusammenführen des Tasking Frameworks mit den einzelnen Komponenten ist sehr aufwendig, da es keine einheitliche Schnittstelle gibt und die Daten der Komponenten und des Tasking Frameworks häufig umkonvertiert werden müssen.

1.2.2 Softwareentwicklung mit dem Tasking Framework

Ein typischer Entwicklungsprozess eines Projekts mit dem Tasking Framework ist in Abbildung 2 dargestellt. Im ersten Schritt setzten sich der Systementwickler und der Spezialist für das Tasking Framework zusammen und erstellen eine Tasking-Umgebung. In dieser Tasking

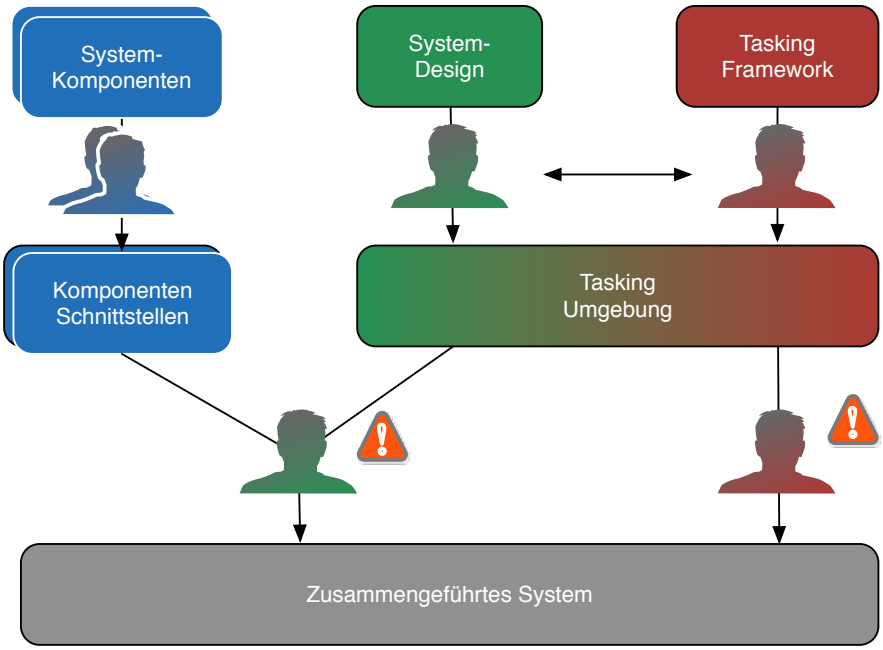


Abbildung 2: Zur Entwicklung an einem Projekt mit dem Tasking Framework ist aktuell neben den Komponenten- und Systementwicklern auch ein Entwickler mit Fachwissen über das Tasking Framework notwendig. Dieser wird auch noch benötigt, wenn ein bestehendes Projekt um eine Komponenten erweitert werden soll. Außerdem muss der System-Entwickler die Komponenten mit dem Tasking Framework zusammenführen und sich dabei in viele Komponenten einarbeiten.

Umgebung werden die Parameter des Tasking Framework, wie zum Beispiel die Priorität der Tasks und die Kommunikationswege mit dem Tasking Framework implementiert. Gleichzeitig stellen die Komponenten-Entwickler für ihre Komponente eine Schnittstelle zur Verfügung. Da die Software-Module von verschiedenen Instituten stammen können, muss der System-Entwickler bei der Zusammenführung verschiedene Code-Stile zusammenbringen. In bisherigen Projekten war dieser Schritt teilweise sehr zweitaufwendig. Sollen im Laufe des Entwicklungsprozesses neue Komponenten eingefügt werden, sind weiterhin Kenntnisse über das Tasking Framework notwendig.

Da die Implementierung der API des Tasking Frameworks viel wiederkehrenden Code enthält und es somit sehr aufwendig ist ein Softwareprojekt auf Basis des Tasking Framework zu erstellen soll dieser Vorgang durch Code-Generierung optimiert werden.

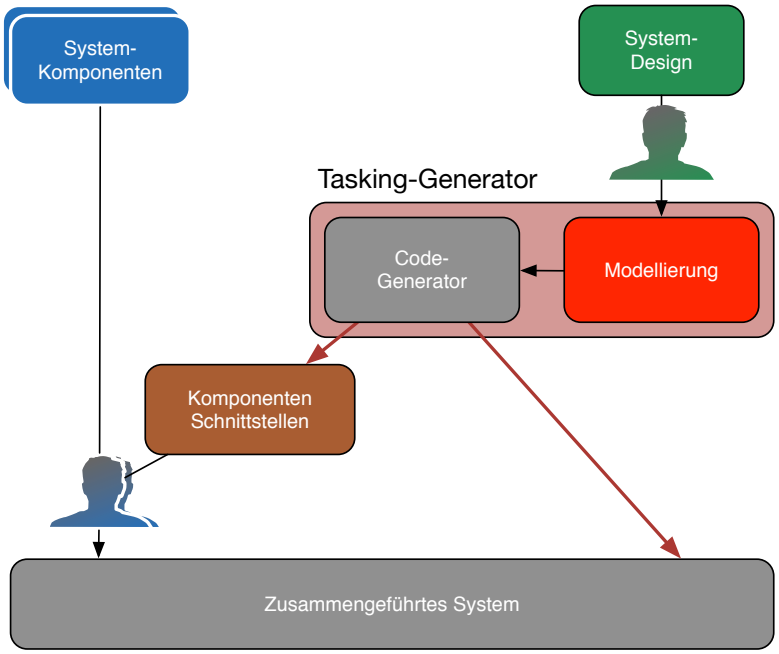


Abbildung 3: Im Gengensatz zu Abbildung 2 wird nun kein Entwickler mit Fachwissen über das Tasking Frameworks mehr benötigt. Der System-Entwickler modelliert das System und kann daraus den notwendigen Code generieren. Zusätzlich werden auch die Schnittstellen des Tasking Frameworks generiert, die dann von den Komponenten-Entwicklern implementiert werden können.

1.2.3 Instanziierung des Tasking Framework mit Hilfe grafischer Modellierungswerkzeuge

Ziel der Code-Generierung ist es, das Know-How über das Tasking Framework in einen Code-Generator zu integrieren, um zur Verwendung keinen Expertenwissen mehr zu benötigen. Der in Abbildung 3 dargestellte Entwicklungsprozess ist zudem deutlich einfacher. Neben der Generierung der missionsspezifischen Komponenten und der Kommunikation des Tasking Frameworks sollen auch global spezifizierte Schnittstellen generiert werden, die dann von den Komponenten-Entwicklern implementiert werden können. Da die Schnittstellen so vereinheitlicht werden, können die Komponenten darauf angepasst werden und das Zusammenführen der Software durch den Systementwickler kann entfallen.

Ein großer Vorteil dieses Ansatzes ist, dass wenn im Verlauf des Projekts neue Komponenten

eingefügt werden sollen, der System-Entwickler das Modell anpassen und den für das Tasking Framework notwendigen Code generieren kann. Auf diese Weise ist es möglich, dass auch im späteren Projektverlauf kein zusätzliches Expertenwissen über das Tasking Framework mehr benötigt wird.

Um den Code für die Implementierung des Tasking Frameworks generieren zu können, soll die Umgebung über eine Domain Specific Language (DSL) beschrieben werden. Da grafische DSLs effektiv die bildliche, menschliche Vorstellungskraft nutzen [AK03, S.38], eignen sich diese besonders für die Modellierung der Kommunikation. Bei der Verwendung einer grafischen DSL, können allerdings keine fertigen Frameworks (vgl. [Eclh]) wie bei der Erstellung von textuellen DSLs verwendet werden.

1.2.3.1 Aufgabenstellung

Die Aufgabenstellung dieser Arbeit ist das Erstellen einer grafischen Modellierungssprache zur Beschreibung des Systemdesigns eines Projekts. Für die Modellierung soll der Modellentwickler kein Wissen über die Implementierung des Tasking Frameworks benötigen. Die Modellierungssprache kann eine Kombination aus bestehenden oder eine neu erstellte sein. Die Entscheidung für eine Sprache bzw. Werkzeuge zum Erstellen einer sollen fundiert begründet werden. Außerdem soll zur Modellierung ein grafischer Editor zur Verfügung stehen, der das Modell validiert, um für die Code-Generierung nur gültige Konstrukte zuzulassen.

1.2.3.2 Rückblick auf bestehende Prototypen

Das Projekt Autonomous Terrain based Optical Navigation (ATON) verwendet Teile des Tasking Frameworks. In diesem Projekt werden hauptsächlich optische Sensoren verwendet, um eine Flugbahn inklusive Landeplatzbewertung auf fremden Himmelskörper autonom und in Echtzeit errechnen zu können [DLRd]. Da das Zusammenführen der Software-Komponenten mit dem Tasking Framework dort besonders aufwendig war, wurde ein Prototyp für einen Code-Generator erstellt, der sowohl die Schnittstellen als auch die Kommunikation der Komponenten generiert [Fra14].

Der Code-Generator des ATON Projekts verwendet ein UML-Model zur Code-Generierung. Die Code-Generierung ist mit dem Eclipse Modeling Framework (EMF) umgesetzt [Fou].

Der Fokus in diesem Projekt lag auf der Code-Generierung und die Modellierung wurde als Konzeptstudie umgesetzt. Diese Arbeit soll auf dem existierenden Code-Generator aufbauen.

1.2.3.3 Ziele im Sinne der Modellierbarkeit

Um mit dem Code-Generator alle Merkmale und Funktionen des Tasking Frameworks abdecken zu können, muss mit der Modellierungssprache die Kommunikation und Ereignissteuerung vollständig beschrieben werden können.

Ziel dieser Arbeit ist die fundierte Entscheidung, mit welchem Ansatz und welchen Werkzeugen die grafische Modellierung umgesetzt werden kann.

Zur Erstellung des Modells soll ein Editor zur Verfügung stehen, der eine grafische Schnittstelle bereitstellt. Mit Hilfe von Diagrammen soll der Nutzer die Software im Editor so beschreiben, dass mit den im Modell gesammelten Daten, der Code-Generator ausgeführt werden kann. Die Bearbeitung des Modells kann entweder mit einem existierenden oder einem neu erstellten Editor erfolgen.

Die Modellierung soll die Benutzung des Tasking Frameworks vereinfachen, um auch von Entwicklern verwendet werden zu können, die die internen Strukturen und Abläufe nicht kennen. Aus diesem Grund soll die Modellierungssprache und der Editor nur für das Tasking Framework gültige Konstrukte zulassen, um im Code-Generator keine undefinierten Zustände zu erzeugen. Dies bedeutet auch, dass der Editor möglichst einfach und übersichtlich gestaltet sein sollte.

Zusätzlich soll der Editor auch eine weitere Validierung des Modells bieten, die den Nutzer auf fehlerhafte Anordnungen oder falsche Diagrammelement-Parameter hinweisen kann.

1.2.3.4 Ziele im Sinne der Code-Generierung

Da wie in Abschnitt 1.2.3.2 beschrieben bereits ein Code-Generator existiert, der Teile der missionsspezifischen Komponenten des Tasking Frameworks generieren kann, ist es Ziel dieser Arbeit, dass die Modellierung auf diesem aufbaut. Da der existierende Generator auf dem Eclipse Modeling Framework beruht, soll die in dieser Arbeit erstelle Modellierungssprache mit dem EMF zusammenarbeiten [Fou].

Ziel dieser Arbeit ist die Machbarkeit der Code-Generierung mit der erstellten Modellierungssprache zu zeigen und den existierenden Code-Generator gegebenenfalls anzupassen. Die Erstellung eines produktiven Code-Generators ist nicht das Ziel, sondern der Fokus liegt auf der Modellierung.

2 Grundlagen der Modellierung und modellgetriebenen Entwicklung

Der Begriff der Modellierung bezeichnet das Arbeiten mit einem Modell [Dr.06a]. Ein Modell bezeichnet entweder ein Vor- oder Abbild eines Objekts von Interesse. Im Modell werden die für den Verwendungszweck relevanten Informationen gesammelt. Eine Beispiel für eine Modell ist eine Modelleisenbahn, die versucht die optischen Eigenschaften der realen Entsprechung verkleinert darzustellen. Für dieses Modell ist die Form und Farbe von Bedeutung, die verwendeten Materialien werden jedoch nicht abgebildet. Im Vergleich zu der Modelleisenbahn die als Abbild verwendet wird, erstellen Architekten von der Errichtung eines geplanten Gebäudes ein Modell. Dieses dient der Präsentation und ist ein Vorbild für das zu entwerfende Gebäude.

Im folgenden Kapitel werden die Grundkonzepte der Modellierung in Bezug auf die Softwareentwicklung beschrieben.

2.1 Modellgetriebene Entwicklung

Die modellgetriebene Softwareentwicklung bezeichnet den Prozess der formalen Beschreibung in einem Modell und der Generierung von Projektinhalten aus diesem [Bro04]. Die modellgetriebene Softwareentwicklung (engl. Model-Driven Software Development), kurz MDSD, ist ein Teil der modellbasierten Entwicklung und verwendet das Modell als zentrales Element, um daraus z.B. Quellcode und Dokumentationen zu generieren. Die Hauptmotivation zur Verwendung der modellgetriebenen Entwicklung ist die Erhöhung der Produktivität [AK03]. Dabei kann die Entwicklung sowohl kurz- als auch langfristig produktiver werden. Durch Generierung aus einem Modell kann Software generiert werden, die schneller mehr Funktionalität bietet als bei manueller Implementierung, da die Funktionen generisch in Templates implementiert werden und dann auf verschiedene Bereiche angewendet werden können. Auf lange Sicht kann einer Änderung der Anforderungen relativ leicht durch Anpassung des Modells entgegnet werden. Da die Software generisch in Form von Templates implementiert wird und die Kontext-spezifischen Inhalte aus dem Modell geladen werden, müssen ähnliche

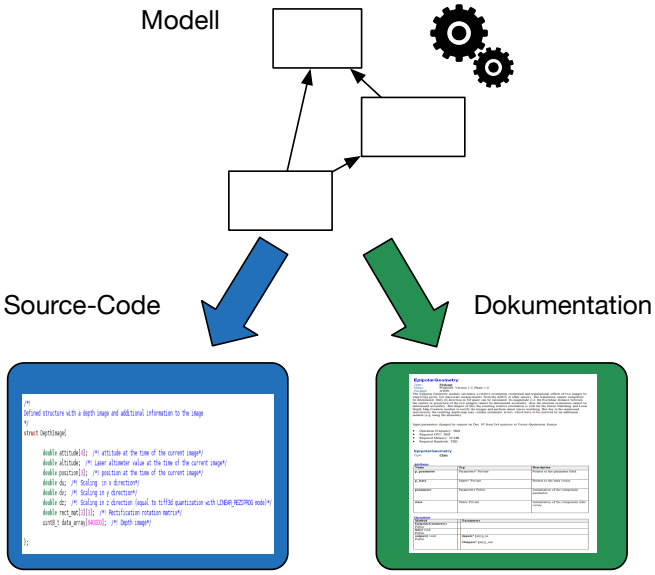


Abbildung 4: Darstellung des Workflows der modellgetriebenen Entwicklung. Begonnen wird dabei immer mit der Erstellung und Beschreibung der Software in einem Modell. Aus dieser formalen Beschreibung können dann z.B. Source-Code und Dokumentationen generiert werden.

Funktionen nicht mehrfach implementiert werden, sondern sie können generiert weden. Um Wiederholungen zu vermeiden, wird die Software dabei plattform- und sprachenunabhängig beschrieben [AW04].

Das Modell dient in der modellgetriebenen Entwicklung also wie das Gebäudemodell des Architekten als Vorbild. Eine Veränderung am Modell führt in der modellgetriebenen Softwareentwicklung jedoch direkt zu einer Änderung der generierten Software. Das Modell dient hier also nicht nur der Präsentation sondern ist essenzieller Bestandteil der Entwicklung. Wie in Abbildung 4 dargestellt beginnt ein Entwicklungsprozess mit der Erstellung eines Modells. Aus dieser formalen Beschreibung können dann Projektinhalte, wie Teil der Software, Unit-Tests oder Dokumentation generiert werden.

Eine Infrastruktur zur modellgetriebenen Softwareentwicklung sollte die von Atkinson und Kühne beschriebenen Anforderungen definieren [AK03, S.37] :

1. Die Modellierungskonzepte die verwendet werden, um das Modell zu erstellen und die dafür definierten Regeln. Ein Beispiel dafür wäre welche Diagrammtypen verwendet

werden.

2. Wie die Modellelemente dargestellt werden
3. und auf welche Weise sie die realen Elemente abbilden.
4. Konzepte, wie die Modellierung erweitert werden kann. Also Methoden zur Anpassung an neue Techniken und Anforderungen.
5. Möglichkeiten wie ein Austausch von Modellierungskonzepten und den Modellen selbst erleichtert werden kann.
6. Möglichkeiten wie die Zuordnung von Nutzer-definierten Abbildungen von Modellelementen zu anderen Projektelementen erleichtert werden kann.

Eine wesentliche Fragestellung die unter anderem mit diesen Konzepten beantwortet werden sollte, ist welche Modellierungssprache verwendet werden soll.

2.1.1 Typen der Metamodellierung

Die Definition einer Modellierungssprache geschieht über die linguistische Metamodellierung [DGD06]. Dabei wird die Sprache von generischen Konzepten einer Modellierungssprache bis zu den realen Elementen schrittweise definiert. In Abbildung 5 sind die vier Ebenen dieser Art der Metamodellierung dargestellt. Die Elemente der realen Welt stellen dabei die unterste Ebene (M0) dieser Illustration da.

Die erste Ebene der Modellierung ist das Nutzermodell (M1). Bei der modellgetriebenen Entwicklung wird darin die formale Beschreibung der zu erstellenden Software durchgeführt. Das Erstellen einer Sprache für dieses Modell ist Ziel dieser Arbeit. Darin soll der Systementwickler die Software beschreiben, um daraus die missionspezifischen Komponenten des Tasking Framework zu generieren.

Die nächste Ebene ist notwendig, um die Sprache des Nutzermodells zu beschreiben. Auf dieser Ebene (M2) werden aus den Grundkonzepten der Modellierung, welche durch die Meta Object Facility (MOF) in Ebene M3 beschrieben sind [Objа], die Sprachen definiert. Die MOF definiert dabei die Konzepte die für die Erstellung einer Sprache notwendig sind. Die oberste Ebene beschreibt dabei also nicht Klassen oder Attribute, sondern nur deren Eigenschaften und Grundkonzepte. Da die Modelle zur Definition der Sprache zwischen dem Nutzer-Modell

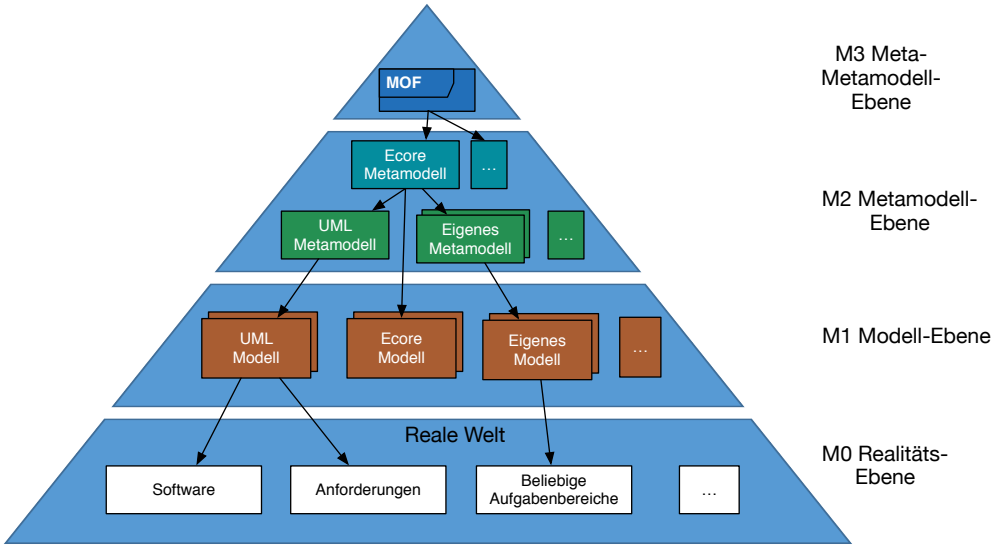


Abbildung 5: Darstellung der linguistischen Metamodellierung. Mit den Konzepten der Meta Object Facility lässt sich ein Metamodell beschreiben. Dieses Metamodell beschreibt das eigentliche Zielmodell, mit dem dann Elemente der realen Welt abgebildet werden können.

und der MOF liegen, werden sie Metamodelle genannt. Metamodelle beschreiben die Elemente des Nutzermodells und deren Zusammenhänge. In Ebene M2 sind sowohl abstrakte als auch für eine Aufgabe speziell erstellte Metamodelle enthalten.

Um die in Abschnitt 2.1 definierten Anforderungen an die Infrastruktur einer modellgetriebenen Entwicklung vollständig umzusetzen, reicht diese eindimensionale Darstellung der Metamodellierung jedoch nicht aus [AK03]. Orthogonal zu der beschriebenen, linguistischen Metamodellierung, gibt es auch die ontologische. Diese Art der Metamodellierung beschreibt „Instanz von“-Beziehungen zwischen Elementen einer linguistischen Metaebene. In Abbildung 6 ist dies anhand eines Beispiels dargestellt. Auf der vertikalen Achse ist dabei die Sprache über linguistische Metamodellierung definiert. Der reale Zug aus Ebene L0 wird durch das Modellelement (L1) abgebildet, welches in dem Metamodell in Ebene L2 definiert wird. Zusätzlich dazu kann auch innerhalb der Ebenen noch klassifiziert werden. Nur so ist es möglich, dass reale Klassen/Arten modelliert werden können. Ein Zug ist ein Transportmittel, der „ICE 38“ ist eine Zug-Instanz. Von der Klasse Zug kann im Nutzermodell sowohl eine Instanz als Objekt, als auch eine Metaklasse erzeugt werden. Falls eine Modellierungssprache

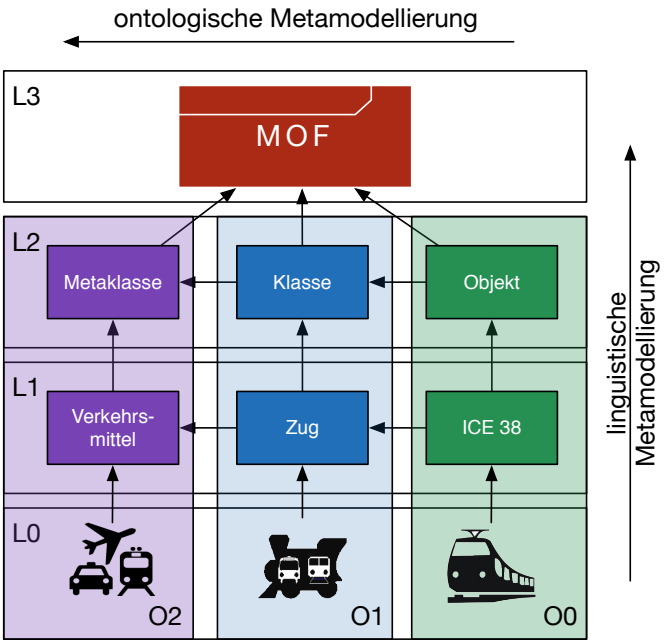


Abbildung 6: Darstellung der zueinander orthogonalen Typen der Metamodellierung. Vertikal sind die linguistischen Ebenen der Sprach-Definition dargestellt, horizontal die ontologischen. So ist der ICE 38 ein Zug, ein Zug wiederum ein Verkehrsmittel.

das Konzept der ontologischen Metamodellierung umsetzt, kann diese über Metaklassen auch nachträglich noch erweitert und angepasst werden.

Die Erstellung einer an ein Problem angepassten Sprache kann also entweder über die Definition einer neuen oder über das Erweitern einer bestehenden Sprache mit ontologischen Konzepten umsetzt werden.

2.1.2 Bedeutung des linguistischen Metamodells

Wie im vorherigen Kapitel beschrieben findet die Definition einer Modellierungssprache über die linguistische Metamodellierung statt. Das Metamodell beschreibt dabei die Syntax und Semantik der Sprache und somit die verwendeten Worte und deren Zusammenhänge [DGD06].

Für die Beschreibung der Bedeutung der Syntax und Semantik einer Sprache wurde auf das Buch „Formal Syntax and Semantics of Programming Languages: A Laboratory Based

Approach“ zurückgegriffen [SK95]:

Syntax. Die Syntax beschreibt die formale Struktur und Form der Wörter einer Sprache und und wie diese miteinander kombiniert werden können, um wohlgeformt (engl. well-formed) zu sein. Die Syntax beschreibt damit nur eine strukturelle Beziehung der Symbole untereinander, ohne auf deren Bedeutung einzugehen.

Semantik. Die Semantik beschäftigt sich mit der Bedeutung syntaktisch korrekter Ausdrücke. Für Computersprachen bedeutet die Semantik das Verhalten des Programms wenn der syntaktisch korrekte Ausdruck ausgeführt wird.

Daraus folgernd bedeutet die Syntax für die Modellierung, welche Elemente an welcher Stelle in ein Modell eingefügt werden können. Die Semantik bedeutet für die modellgetriebene Softwareentwicklung, welcher Code aus einem validen Konstrukt der Modellierungssprache erstellt wird.

Aufbauend auf dem oben genannten Buch beschreibt Prof. Dr. Dominikus noch weiter den Zusammenhang zwischen Syntax und Semantik [Pro08]. Die Sprache kann syntaktisch korrekte Ausdrücke enthalten die semantisch nicht korrekt sind. Je konkreter die Syntax einer Sprache ist, umso weniger muss durch Semantik überprüft werden. In der Modellierung könnte man als Beispiel ein Modellelement betrachten, dass als Attribut einen Typ enthält. Bei einer syntaktischen Beschreibung, bei der das Element beliebige Klassen enthalten kann, ist es semantisch inkorrekt, wenn man eine abstrakte Klasse als Typ angibt. Bei einer syntaktischen Beschreibung, die nur nicht-abstrakte Klassen zulässt, würde die semantische Überprüfung nach abstrakten Klassen wegfallen.

Die Syntax einer Modellierungssprache kann in abstrakt und konkret aufgeteilt werden [CGS12]. Die abstrakte Syntax sind die Modellierungskonzepte, also die Art wie ein Modell aufgebaut wird. Die konkrete Syntax beschreibt die abstrakte Syntax anhand konkreter Modellelemente.

Das Vorgehen zur Definition einer Modellierungssprache ist die Beschreibung der konkreten Syntax über eine Metasprache und anschließend der Erstellung von inhaltlichen Regeln zur Beschreibung der Semantik [Com12]. Eine Metasprache implementiert die Konzepte des MOF und definiert sich selbst. Die Grundelemente einer solchen Metasprache sind in der

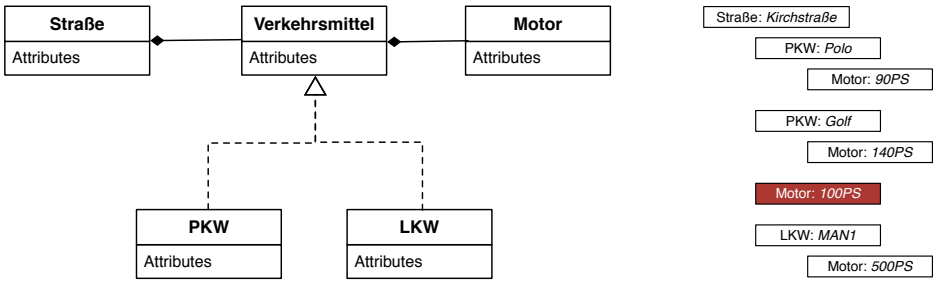


Abbildung 7: Darstellung eins Beispiels zur konkreten Syntax. Links das Metamodell mit dem die rechts verwendete Sprache definiert wird. Die Modellelemente PKW und LKW erben von dem abstrakten Element Verkehrsmittel. Der rechte Ausdruck ist syntaktisch falsch, da ein Motor kein direktes Kindelement der Straße sein darf.

Regel Klassen und Attribute. Die Modellelemente sind als Klassen umgesetzt, die wie in Abbildung 7 dargestellt voneinander erben können. Außerdem enthalten sie Attribute die auf andere Modellelemente referenzieren können. Über die Attribute und Referenzen eines Elements kann beschrieben werden, wie diese miteinander kombiniert werden können. Auf der rechten Seite der Abbildung 7 ist ein syntaktisch inkorrekter Ausdruck gezeigt. Die Definition im Metamodell sieht vor, dass Motoren nur in Verkehrsmittel enthalten sein dürfen, nicht als direktes Kindelement der Straße.

Das Metamodell kann zusätzlich um Semantik erweitert werden, um wohlgeformten Ausdrücke auf die Bedeutung und den Sinn der Elementkombinationen zu überprüfen. Die Bedeutung der Modellelemente kann in Rahmenbedingungen (engl. Constraints) ausgedrückt werden, die den Elementen jeweils zugeordnet werden.

Das linguistische Metamodell beschreibt also sowohl die Syntax als auch die Semantik einer Sprache. Als Ergebnis erhält man die Modellelemente des Nutzermodells, welche dann in Dateien serialisiert werden können.

2.1.3 Serialisierung und Instanziierung des Modells

Um die Informationen der Modelle speichern und verwenden zu können, müssen sie serialisiert werden. Modelle der modellgetriebenen Softwareentwicklung verwenden in der Regel das XML Metadata Interchange (XMI) Format um die Daten in Dateien zu speichern. XMI ist

Abbildung 8: Darstellung der Instanziierung eines Modells. Links das Metamodell einer trivialen Sprache mit nur drei Elementen. Die mittlere Darstellung zeigt ein Modell dieser Sprache im XMI-Format. Rechts dann die grafische Darstellung dieses Modells.

ein Standard der Object Management Group [Objd]. Das Austauschformat kann für alle Metadaten verwendet werden, dessen Metamodell mit den Konzepten der MOF ausgedrückt werden kann [LLLZ10]. Mit XMI als Standard für die Serialisierung ist es möglich Modelle verschiedener Sprachen ineinander zu transformieren.

XMI basiert auf der Extensible Markup Language (XML) welche ein offener Standard des World Wide Web Consortium (W3C) ist [W3C]. XML ist plattform- und sprachenunabhängig und kann auch unabhängig von Softwareentwicklung verwendet werden.

Da XMI ein offener Standard ist, gibt es mehrere Implementierungen davon. Am bekanntesten ist das Eclipse Modeling Framework [Fou]. Die Werkzeuge, die das XMI-Format implementieren, stellen in der Regel Metasprachen zur Verfügung. Diese bauen auf den Konzepten des XMI auf und dienen der Definition von Metamodellen. Beispiele für Metasprachen sind z.B. Ecore von dem EMF oder die Graph-Object-Property-Port-Role-Relationship(GOPPRR) Sprache der MetaEdit+ Modellierungsumgebung [DvIL12]. Für die im Metamodell definierten Elemente werden dann durch die Modellierungsumgebung Softwareentsprechungen bereitgestellt, um programmatisch auf die Modelle zugreifen zu können.

Aufbauend auf den Werkzeugen zur Instanziierung der Modelle im Code können die Elemente auch grafisch dargestellt werden. In Abbildung 8 ist der programmatische Zugriff auf ein XMI-Modell und die Darstellung mit Hilfe des Graphical Modelling Frameworks (GMF) dargestellt [Eclg].

2.2 Grafische Modellierungssprachen

Auf Grund der großen Popularität von grafischen Modellen für sowohl Dokumentation und Präsentation als auch modellbasierter Entwicklung gibt es zahlreiche existierende Modellierungssprachen. Im folgenden Abschnitt werden die für das Anwendungsgebiet der eingebetteten Raumfahrtsoftware relevantetsten Sprachen kurz vorgestellt. So ist die, für die Modellierung von Software, wohl am meist verbreitetste Sprache die Unified Modeling Language (UML). Die Systems Modeling Language (SysML) als Erweiterung von UML ermöglicht durch einen abstrakteren Blick auf das Gesamtsystem die Verwendung in dem Systems Engineering. Von NASA und ESA wird außerdem die Sprache Architecture Analysis and Design Language (AADL) für die Modellierung genutzt.

2.2.1 Unified Modeling Language

Die Unified Modeling Language (engl. für vereinheitlichte Modellierungssprache), kurz UML, ist eine allgemeine Modellierungssprache im Kontext der Softwareentwicklung. Die Sprache wurde 1994-95 entwickelt und 1997 von der Object Management Group veröffentlicht [Objb]. Seit 2005 werden die aktuellsten Versionen außerdem von der International Organization for Standardization (ISO) standardisiert.

Zur allgemeinen Einarbeitung in die Sprache wurde das Buch „UML2 Glasklar“ verwendet [RQZ12]. UML dient der Beschreibung von sowohl dynamischen als auch statischen Aspekten einer Software. UML definiert dafür Bezeichner, die auf den Konzepten des XMI beruhen und legt deren Beziehungen untereinander fest. Für UML gibt es auch eine auf dem EMF basierende Implementierung in der Ecore-Metasprache [Com12]. Unter anderem auf Basis dieser gibt es Editoren, die eine Bearbeitung von UML-Modellen ermöglichen.

Neben den Elementen der Sprache definiert UML außerdem grafische Notationen zu den definierten Elementen und stellt Diagramme bereit, in denen diese verwendet werden. Dabei gib es Diagramme zur Beschreibung der Struktur und des Verhaltens einer Software. Die Struktur wird zum Beispiel über Klassen-, Komponenten- oder Verteilungsdiagramme beschrieben, für die Beschreibung des Verhaltens gibt es unter anderem Aktivitäts-, Sequenz- oder Zustandsdiagramme.

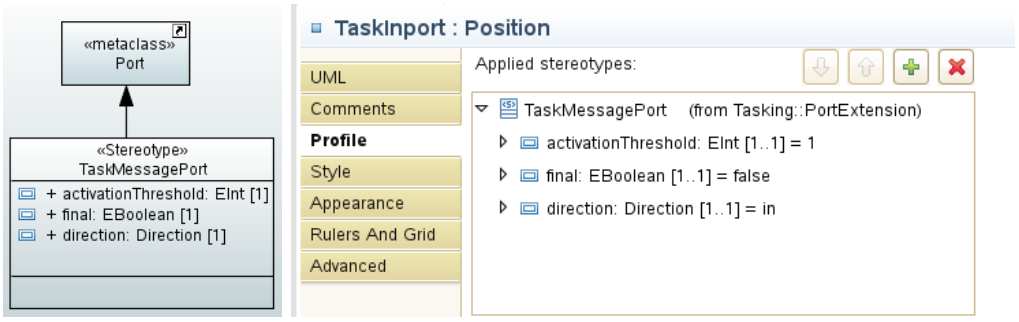


Abbildung 9: Links ist die Definition eines UML Profils dargestellt. Die Metaklasse „Port“ wird durch den Stereotyp „TaskMessagePort“ erweitert. Rechts sieht man die Eigenschaften einer Instanz des neuen Elements im Modell. Das UML-Element, welches auf einem einfachen Port basiert, besitzt nun zusätzlich die links definierten Attribute des Stereotyps als Elementeigenschaften.

Object Constraint Language(OCL). UML beinhaltet eine textuelle Sprache mit der Einschränkungen an Elemente beschrieben werden können. OCL kann von den gängigen UML-Editoren erkannt und interpretiert werden.

UML Profile. Neben diesen Diagrammen zur Beschreibung der Software gibt es auch UML Profil Diagramme. UML Profile setzten das Konzept der Erweiterbarkeit einer Sprache um [FFVM04]. Die vierte der in Abbschnit 2.1 beschriebenen Anforderungen an eine modellgetriebene Softwareentwicklung wird somit durch die Verwendung von UML automatisch erfüllt. Ab UML2 ist diese Erweiterbarkeit durch ontologische Konzepte umgesetzt und ermöglicht das Anpassen der Sprache ohne Bearbeitung des Metamodells. Dies bietet für UML zahlreiche Vorteile, da viele Editoren ein festes Metamodell verwenden, welches nicht bearbeitet werden kann. Das Basis-Konzept von UML Profilen ist die Erstellung und Erweiterung einer ontologischen Metaklasse. Diese Metaklasse ist in dem UML-Metamodell enthalten und ermöglicht es beliebigen Elemente um Attribute zu erweitern. Dafür werden diese in Stereotypen eingebaut, welche dann mit den Metaklassen verbunden werden. Durch Metaklassen erstellte Erweiterungen stehen dann wie in Abbildung 10 gezeigt im Modell inklusive der erweiterten Attribute zur Verfügung. Die neuen Elemente des UML Modells können zusätzlich mit Icons, also grafischen Notationen, angepasst werden.

Neben diesen Erweiterungen ist es auch möglich Bedingungen an neue Element zu stellen und die Sprache somit einzuschränken. Eine Kombination der in OCL geschriebenen Bedingungen

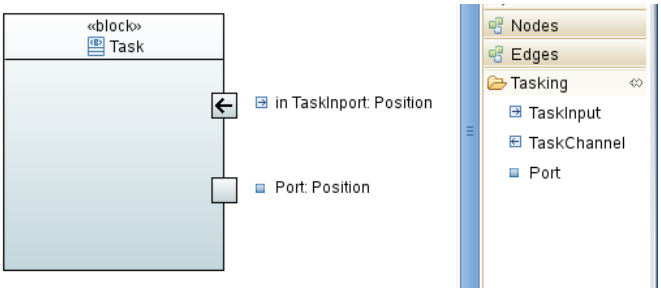


Abbildung 10: Darstellung der Erweiterung eines UML Komponentendiagramms, durch die in Abbildung 9 definierten Modell-Elemente. Die erweiterten Elemente können wie normale UML-Elemente auch in Diagrammen verwendet und die Darstellung durch eigene Icons angepasst werden.

und der Erweiterung um neue Elemente durch UML Profile ermöglicht so das formale Anpassen der Semantik von UML.

2.2.2 Systems Modeling Language

Die Sprache Systems Modeling Language (SysML) ist eine über ein UML-Profil umgesetzte Erweiterung von UML. Wie auch UML ist SysML von der OMG standardisiert [Objc]. Wie der Name schon sagt richtet sich ihr Fokus auf die Beschreibung von Systemen, nicht von Software. Neben der Erweiterung um neue Elemente und semantischen Anpassungen wurden für SysML auch neue Diagramme mit eigener Syntax definiert [IKS13].

Mit einem neuen Diagramm in SysML ist die Beschreibung von Anforderungen möglich. Eine Anforderung (engl. Requirement) hat dabei einen Namen, einen Anforderungstext und eine eindeutige ID. In dem Anforderungsdiagramm wird die Beziehung der Anforderungen untereinander beschrieben. Neben diesen neuen Möglichkeiten Anforderungen zu modellieren, wird mit SysML ein abstrakterer Blick auf das System ermöglicht. So werden Klassen nicht mehr verwendet und es gibt keine Klassendiagramme mehr. Das Hauptkonzept in SysML ist der als Metaklasse umgesetzte Block. Der Block steht für einen Teil des Systems und kann sowohl Software als auch Hardware enthalten. Die Entsprechung des Klassendiagramms in SysML ist das Blockdiagramm. Neben der abstrakteren Sicht auf die Elemente bietet ein Blockdefinitionsdiagramm gegenüber einem Klassendiagramm noch weitere Vorteile, so können mit den Blöcken nicht nur strukturelle Eigenschaften sondern auch Verhalten

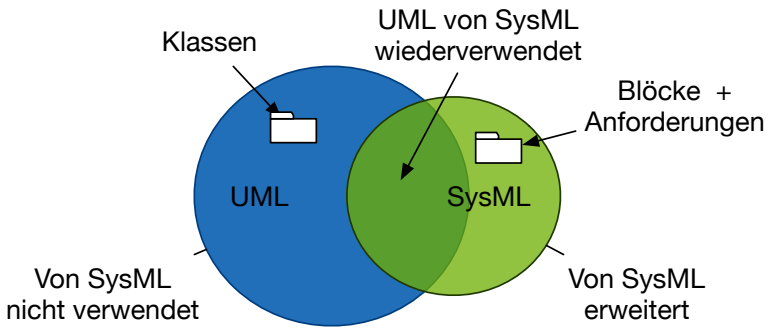


Abbildung 11: Darstellung des Zusammenhangs von UML und SysML. SysML erweitert UML um neue Elemente und Diagramme und verwendet nur noch ein Teil der eigentlichen UML-Elemente. Die nicht mehr verwendeten sind im Metamodell noch enthalten, haben in den neuen Diagrammen keine Bedeutung mehr.

modelliert werden.

Mit Hilfe von Constraint-Blocks können Regeln an ein System beschrieben werden die erfüllt werden müssen. Die Constraints können dabei in OCL erstellt werden. Die Blöcke können Parameter enthalten die von den Constraints verwendet werden. Auf diese Weise lassen sich dynamische Werte, die sich über die Parameter errechnen lassen, darstellen. Solche parametrisierten Blocke können in Zusicherungsdiagrammen verwendet werden, welche eine Erweiterung von einfachen Internen Blockdiagrammen darstellen.

2.2.3 Architecture Analysis and Design Language

AADL ist eine von der Society of Automotive Engineers (SAE) standardisierte Sprache zur Beschreibung von Software- und Hardwarearchitektur. Der Fokus der Sprache liegt auf eingebetteten Echtzeitsystemen [Mun14]. Im Vergleich zu SysML wird weniger das Gesamtbild beschrieben, sondern mehr die technischen Details. Dafür können in einem Modell Elemente zur Beschreibung von sowohl Soft- als auch Hardware eingefügt werden. Ein AADL-Model basiert auf Komponenten, die in Software und Execution Plattform (engl. für Ausführungsplattform) getrennt werden. Zur Beschreibung von Software gibt es Elemente wie Daten, Threads oder Prozesse. Die Hardware wird mit Elementen wie Bus, Memory oder Prozessor beschrieben. Ein typisches AADL Diagramm ist in Abbildung 12 dargestellt.

Wie UML auch bietet AADL Erweiterungsmechanismen. Modellelemente können wie bei

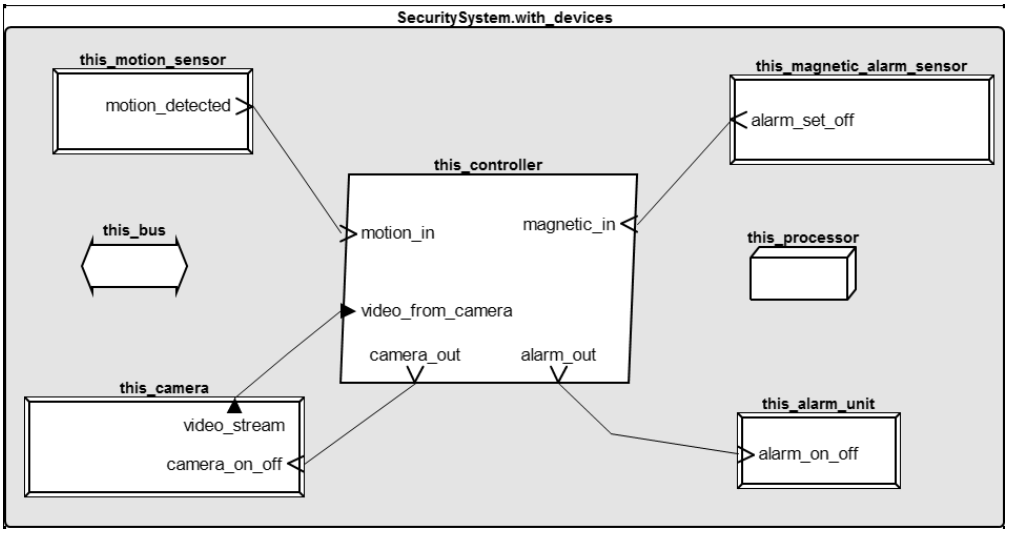


Abbildung 12: Darstellung eines AADL Diagramms der AADL Tutorial Seite [Tut]. Mit AADL ist es möglich die Kommunikation verschiedener Komponenten untereinander zu modulieren. In diesem Diagramm ist Beschreibung allerdings sehr technisch. Linien entsprechen Bussen, Diagramm-Elemente sind Sensoren, Prozessoren oder Speicherelemente.

der Vererbung von Programmiersprachen um neue Attribute erweitert werden. Neben dem einfachen Erweitern von Elementen lässt sich AADL auch über Annex-Sprachen erweitern. Über solche Erweiterungssprachen lassen sich in AADL zum Beispiel Constraints einfügen. Mit AADL selbst lassen sich Regeln zu den Kernelementen der Sprache definieren, für erweiterte Elemente wird die Annex-Sprache Requirements and En- forcements Analysis Language (REAL) benötigt [GH10]. Neben dieser Erweiterungssprache gibt es auch Sprachen zur Modellierung von Fehlern und zur Beschreibung des Verhaltens der Elemente.

2.3 Verwandte Arbeiten

Auf Grund der großen Verbreitung der modellgetriebene Softwareentwicklung gibt es bereits zahlreiche Werkzeuge zur Code-Generierung aus Modellen. Im folgenden wird eine Werkzeugkette der ESA zur Code-Generierung aus AADL Modellen und eine Möglichkeit zur Erstellung grafische DSLs vorgestellt.

2.3.1 TASTE

Mit TASTE wird von der ESA eine Open-Source Werkzeugkette zur Entwicklung von eingebetteten Echtzeitsystemen entwickelt [PCD12]. Der Name steht für „The ASSERT Set of Tools for Engineering“ und beschreibt die Philosophie, den Nutzer funktionalen Code in einer ihm überlassenen Sprache schreiben zu lassen und die Komponenten dann automatisiert zusammenführen zu lassen. Die Hauptaspekte werden dabei mit den Modellierungssprachen AADL und ASN.1 modelliert. Die Werkzeugkette liefert dabei fertige Binaries, die auf die Charakteristika einer eingebetteten Sprache optimiert sind. Der generierte Code benötigt nur geringe Ressourcen, es können Echtzeit-Anforderungen definiert werden und der Code kann mit Hardware kommunizieren. TASTE ist somit eine sehr gute Lösung zur genauen Beschreibung der Hardware und der daraus möglich werdenden Kombinierung verschiedene Softwarekomponenten aus verschiedener Sprachen.

Unabhängig davon das der Code-Generator in Python und Ada implementiert und somit nicht mit dem EMF kompatibel ist [LZPH09], kann TASTE nicht zur Generierung des Tasking Frameworks verwendet werden. TASTE verwendet eigene Frameworks zur Kombinierung von Softwarekomponenten. Ein Werkzeug zur modellgetriebenen Entwicklung mit dem Tasking Framework und TASTE stellen damit beide Möglichkeiten zur Kombinierung von Softwareelementen zur Verfügung, haben jedoch leicht unterschiedliche Anwendungsgebiete. Die Werkzeugkette der ESA wird entwickelt um hardwarenahen Code durch eine detailreiche Beschreibung der Hardware im Modell zu ermöglichen. Das in dieser Arbeit beschriebene Werkzeug hingegen soll keinen hardwarenahen Code selbst generieren, sondern die Komponenten eines ereignisgesteuerten Echtzeit-Laufzeitsystem zusammenführen. Im Vergleich zu TASTE liegt der Fokus also nicht auf der Generierung von hardwarenahem Code sondern auf der großen Konfigurierbarkeit der Ereignissteuerung des Tasking Framework. Ein Werkzeug zur Modellierung des Tasking Framework benötigt somit einen abstrakteren Blick auf die Software als es bei TASTE der Fall ist. Da das Tasking Framework in Zukunft ebenfalls in ein Framework zur Kombination verschiedener Programmiersprachen, insbesondere Ada, weiterentwickelt werden soll, macht eine Auseinandersetzung mit TASTE besonders bei der Erstellung des Code-Generators Sinn.

Im Sinne der Modellierung sieht man an TASTE, dass es mit AADL möglich ist die Beziehungen und Kommunikation der Komponenten untereinander zu beschrieben (siehe Abbildung 12). Da TASTE auf Grund des Code-Generators ausgeschlossen werden kann, bietet sich für das

Tasking Framework allerdings eine abstraktere, an Software orientiertere Sprache an, die an die Parameter der Ereignissteuerung des Tasking Frameworks angepasst werden kann.

2.3.2 Grafische Domain-specific Language mit UML

Eine Möglichkeit zur Erstellung einer an eine Aufgabe angepasste Sprache mit UML wird von Rybola und Richta beschrieben [RR12]. Um die Transformation von Modell zu Code effektiv umzusetzen, werden dabei zuerst eigene, an die Aufgabe angepasste Sprachelemente mit UML-Profilen erstellt. Da bei der Modellierung Fehler entstehen können, werden neben diesen angepassten Elementen auch Regeln an die Elemente erstellt. Die in OCL erstellten Regeln beschreiben wie das Profil verwendet werden muss. In Diagrammen können die Beziehungen der erweiterten Elemente untereinander beschrieben werden. Eine falsche Verwendung kann über in OCL definierte Regeln verhindert werden. Neben der manuellen Erstellung von solchen Constraints können aus Diagrammen zur Beziehung der erweiterten Elemente und deren Regeln die auch Constraints generiert werden.

Eine Implementierung der auf UML Profilen und OCL basierten Anpassung einer Sprache an eine Aufgabe kann z.B. mit dem im EMF umgesetzten UML Editor Papyrus realisiert werden [GDTS10]. Eine Zusammenführung mit dem existierenden Generator wäre so ohne großen Aufwand möglich.

3 Konzepte

Das Ziel der Anwendung der modellgetriebenen Entwicklung ist die Steigerung der Produktivität von Projekten welche das Tasking Framework benutzen. Wie in Abbildung 13 dargestellt soll die Software dabei mit mehreren Diagrammen beschrieben werden. Die in den Diagrammen enthaltenen Informationen werden dann in einem Modell gesammelt. Aus dem Modell können dann sowohl Source-Code als auch Dokumentationen und Tests generiert werden. Der Fokus dieser Arbeit liegt auf dem ersten Schritt dieses Arbeitsablaufs, der Modellierung zur Beschreibung der Software. Es wird nicht die Erstellung eines fertigen Code-Generators beschrieben. Eine Auseinandersetzung mit dem zu generierenden Code ist jedoch unerlässlich. Nur über eine Analyse des Verwendungszwecks des Modells ist die Erstellung effektiver Werkzeuge zur Modellierung möglich.

Im folgenden Kapitel wird das Tasking Framework analysiert, um geeignete Modellierungstechniken zu finden. Außerdem wird auf die Anforderung an die Editoren und der modellgetriebenen Entwicklung eingegangen. Aus diesen Anforderungen wird dann das Konzept einer Modellierungssprache und deren Editoren hergeleitet.

3.1 Analyse des Tasking Framework

Zur Optimierung der Entwicklung von Projekten, die auf dem Tasking Framework aufbauen, soll die Kommunikation und deren Schnittstellen generiert werden. Um diesen Quellcode generieren zu können, müssen alle für die Implementierung wichtigen Details im Modell enthalten sein. Außerdem muss die Ereignissteuerung in Diagrammen beschrieben werden können.

3.1.1 Elemente des Tasking Framework

Folgend sind die wichtigsten Elemente des Tasking Frameworks und ihre Aufgaben beschrieben [Mai], diese sollen grafisch abgebildet werden können:

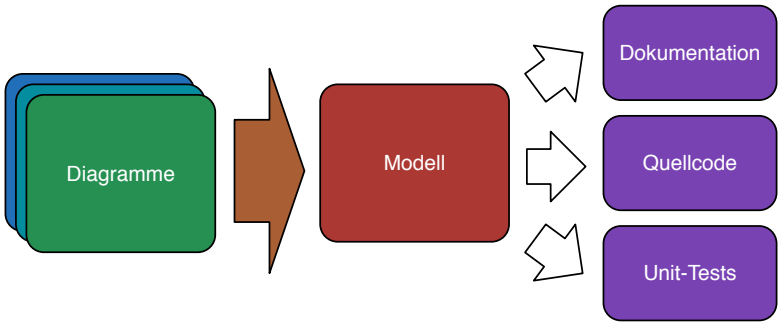


Abbildung 13: Überblick über den Informationsfluss in der modellgetriebenen Entwicklung. Die Software wird in verschiedenen Diagrammen beschrieben und die so gesammelten Informationen im Modell gespeichert. Aus dem Modell werden dann Projektartefakte wie Quellcode oder Dokumentationen generiert.

Task. Eine Aufgabe oder Software-Komponente, die mit dem Tasking Framework parallelisiert ausgeführt werden soll. Tasks haben verschiedene Parameter, wie eine Priorität, die im Modell enthalten sein müssen. Zudem sollen ein- und ausgehende Daten der Module beschrieben werden können.

TaskSet. Die Tasks können in verschiedenen Threadpools ausgeführt werden. Ein TaskSet ist ein solcher Threadpool. Verschiedene TaskSets können getrennt voneinander zurückgesetzt, neugestartet oder pausiert werden. Die Modellierung soll beschreiben, welche Tasks zu welchen TaskSets gehören. Zur Erstellung eines TaskSets im Code muss die Anzahl an Tasks, die in diesem ausgeführt werden sollen angegeben werden. Für die Code-Generierung sollte diese Information in den TaskSet-Elementen des Modells enthalten sein.

TaskChannel. Die Kommunikationskanäle zwischen den Tasks heißen TaskChannel. Hauptaufgabe der Channels ist die Benachrichtigung aller lesenden Teilnehmer sobald ein Ereignis oder Daten in einen Channel geschrieben werden. Die Kommunikation kann Nutzdaten enthalten, muss aber nicht. Wie in Abbildung 14 dargestellt ermöglichen TaskChannels eine n:m Kommunikation. Es können theoretisch beliebig viele Tasks in einen Channel schreiben und daraus lesen. Auf Grund der interne Datenstrukturen kann es jedoch Channel-Typen geben, die eine begrenzte Anzahl an lesenden oder schreibenden Tasks zulassen. Das Tasking Framework bietet vier Basis-Channel an, zwischen den im Modell ausgewählt werden können

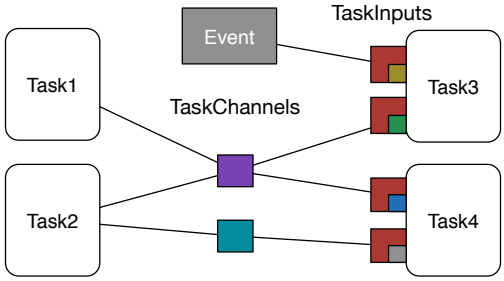


Abbildung 14: Darstellung der Kommunikations-Komponenten des Tasking Frameworks. Tasks können Daten in TaskChannels schreiben und daraus lesen. Über TaskInputs werden Tasks beim Eintreffen neuer Daten benachrichtigt. Zusätzlich können auch TaskEvents Signale an TaskInputs senden.

soll.

- Trigger-Signal: Enthält keine Daten
- First-In-First-Out-Speicher
- Last-In-First-Out
- Zwei Buffer, bei denen jeweils immer der aktuelle ausgelesen wird

TaskEvent. Neben TaskChannels können auch TaskEvents Ereignissignale senden. Im Unterschied zu TaskChannels senden TaskEvents die Signale jedoch nicht, wenn ein vorheriges Ereignis eingetreten ist, sondern periodisch. Die Tasks können so nicht nur Datenstrom-gesteuert sondern auch zu bestimmten Zeiten ausgeführt werden.

TaskInput. TaskInputs sind die Empfänger des Benachrichtigungssystems. Pro Benachrichtigungsquelle können so Parameter angegeben werden. Ein TaskInput wird aktiviert sobald eine bestimmte Anzahl an Signalen eingegangen ist. Ein Task wird immer dann ausgeführt, wenn alle TaskInputs aktiviert sind.

Neben diesen Elementen des Tasking Frameworks sollen auch die von den TaskChannels verwendeten Datentypen und die Parameter der Softwaremodule generiert werden können.

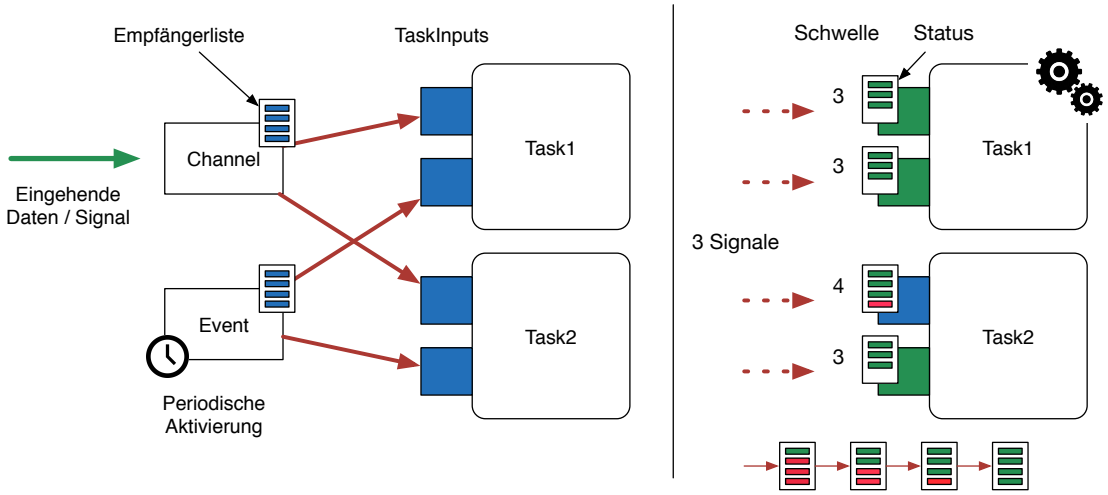


Abbildung 15: Darstellung der Ereignissteuerung des Tasking Frameworks. Entweder periodisch (TaskEvents) oder falls Daten eingehen (TaskChannels) werden Benachrichtigungen an die Empfänger gesendet. Die Empfänger-Tasks werden erst ausgeführt, sobald die Schwelle aller TaskInputs erreicht ist.

3.1.2 Ereignissteuerung des Tasking Framework

Die Softwaremodule, die das Tasking Framework verbindet, werden ereignisgesteuert ausgeführt. Ein Ereigniss kann entweder durch die Übertragung von Daten oder zeitlich ausgelöst werden. In Abbildung 15 sind mögliche Kombinationen der Kommunikations- und Ereignissteuerung des Tasking Frameworks dargestellt. TaskChannels und Events haben eine Liste an Empfängern die sie benachrichtigen, sobald ein Ereignis ausgelöst wird. TaskInputs als Empfänger dieser Benachrichtigungen haben eine Schwelle, die angibt, ab wie vielen Signalen ein TaskInput aktiviert wird. Sobald alle TaskInputs aktiviert sind wird der Task ausgeführt. In Abbildung 15 wird rechts der zweite Task noch nicht ausgeführt, da bei dessen erstem TaskInput die Schwelle von 4 noch nicht erreicht wurde.

Bei einer Schwelle von „0“ ist der Eingang optional, der Task wird also auch ausgeführt, falls der TaskInput nicht aktiviert wurde. Finale TaskInputs führen bei einer Aktivierung zu einer sofortigen Ausführung des Tasks unabhängig von den anderen TaskInputs.

3.1.3 Sonstige Anforderungen

Neben diesen funktionalen Anforderungen zur Modellierung des Tasking Framework gibt es auch Infrastruktur- und Plattformanforderungen an den Editor. Da das Tasking Framework und Software die darauf basieren unter Linux entwickelt werden, ist es notwendig, dass die Modellierung auch auf diesem Betriebssystem vorgenommen werden kann.

Der Editor sollte die modellierten Informationen in einem Modell im XMI-Format speichern, um möglichst generisch mit dem Code-Generator darauf zugreifen zu können. Die Diagramm-Editoren sollen nur gültige Konstrukte der Elemente des Tasking Frameworks bzw. deren Repräsentation im Diagramm zulassen. Eine direkte Verbindung der Software-Module untereinander soll z.B. nicht möglich sein. Außerdem soll der Editor die Parameter im Diagramm validieren können, um Eingaben, die entweder im Code-Generator oder im generierten Tasking Framework Fehler auslösen, zu verhindern. Eine solche Validierung sollte möglichst einfach und generisch möglich und leicht erweiterbar sein.

Das Tasking Framework ist in C++ implementiert. Die Tasks einer Software, die das Tasking Framework verwendet, sollten jedoch auch in Ada oder mit Matlab erstellt werden können. Um mit dem Code-Generator eine angepasste API zu erstellen, sollte in dem Editor für die Tasks angegeben werden können, auf welche Weise diese implementiert werden sollen.

3.2 Auswahl relevanter Modellierungskonzepte

Der erste Schritt zur Erstellung einer Entwicklungsumgebung, die eine Modellierung zulässt, ist die Auswahl relevanter Modellierungskonzepte. Dabei muss entschieden werden, auf welche Weise die Beschreibung der Software stattfinden soll. In Abschnitt 2.1 sind die Anforderungen aufgeführt, die eine modellgetriebene Entwicklungsumgebung beschreiben muss.

Konzepte zur Erstellung des Modells. Die erste zu definierende Eigenschaft der Entwicklungsumgebung sind die Konzepte und Regeln zur Erstellung des Modells. Um die menschliche, grafische Vorstellungskraft auszunutzen, soll das Modell aus Diagrammen erstellt werden [AK03, S.38]. Da das Tasking Framework verschiedene Softwaremodule kombinieren und deren Kommunikation untereinander regeln kann, ist eine Beschreibung der allgemeinen Softwarestruktur und der Kommunikationskanäle wichtig. Der effektivste Weg zur Beschreibung

dieser Aspekte ist ein Block- oder Komponentendiagramm, in dem jeweils die Softwaremodule als Element dargestellt und dann verbunden werden. Zur Beschreibung der Datenstrukturen, die ausgetauscht werden, ist ein Klassendiagramm sinnvoll. Eine Beschreibung dieser beiden Aspekte setzt also zwei verschieden Abstraktionsebenen voraus. Die Datentypen werden auf Basis von Software-Klassen beschrieben, die Kommunikation der Software-Module benötigt eine Systemsicht.

Neben der Beschreibung der Kommunikation und Systemstruktur muss auch die Verteilung der Tasks modelliert werden können. Eine Beschreibung wie in der rechten Darstellung der Abbildung 16 ermöglicht eine einfache Zuordnung von Tasks zu TaskSets.

Darstellung der Modellelemente. Anforderungen 2 und 3 der modellgetriebenen Entwicklung aus Abschnitt 2.1 beschreiben, dass definiert werden muss, wie ein Element im Modell enthalten ist und wie die Modellelemente die realen Elemente repräsentieren. Um eine gute Notation der Elemente im Modell zu finden sollte beachtet werden, wie diese Elemente in der realen Welt verwendet werden. Die Tasks des Tasking Frameworks sind auch im Modell die zentralen Elemente. Sie beschreiben Software-Komponenten die mit dem Tasking Framework ausgeführt werden sollen. Über Verbindungen zwischen diesen werden die Kommunikationswege beschrieben. Elemente zur Ereignissteuerung wie die der TaskEvents oder TaskInputs haben geringere Bedeutung und werden kleiner modelliert. Parameter der Module und Datentypen entsprechen realen Klassen und werden als solche im Diagramm modelliert. Eine endgültige Notation der Elemente setzt jedoch die Auswahl der Modellierungssprache voraus.

Konzepte für Nutzerweiterungen. Der vierte zu definierende Punkt einer modellgetriebenen Entwicklung sieht Konzepte für Nutzererweiterungen vor. Das Grundkonzept der Modellierung der Software des Tasking Framework beruht darauf möglichst einfach zu sein und keine undefinierten Zustände im Code-Generator zuzulassen. Ein Konzept zur Erweiterung der Sprache muss also verhindern, dass beliebige Elemente verändert werden können. Die Werkzeuge zur modellgetriebenen Entwicklung des Tasking Frameworks werden mit diesem weiterentwickelt und stellen so in der Regel alle notwendige Funktionalität zur Verfügung. Nutzererweiterungen machen jedoch Sinn für die Anpassung von TaskChannels und Softwarekomponenten, die durch angepasste Implementierungen zusätzliche Parameter besitzen können. Für die Implementierung eines eignen Datenspeichers könnten so z.B. zusätzliche

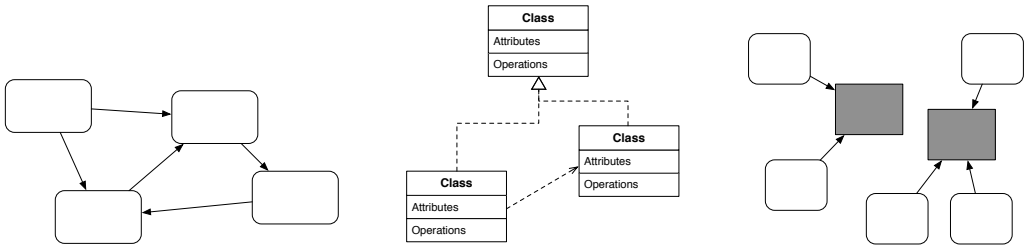


Abbildung 16: Darstellung notwendiger Diagrammtypen zur Beschreibung der auf dem Tasking Framework aufbauenden Software. Links ein Komponenten/Block-Diagramm, welches die Kommunikation beschreibt. In der Mitte ein Klassendiagramm zur Beschreibung der Datenstrukturen und Parameter. Rechts ein Verteilungsdiagramm zu Beschreibung der Aufteilung der Tasks auf die TaskSets.

Parameter wie eine Länge einer Liste benötigt werden. Im Editor muss dafür ein Möglichkeit bestehen wie eigene Elemente erstellt und konfiguriert werden können. Der Code-Generator kann für eine solche Erweiterung Klassen-Rümpfe generieren, die dann vom Nutzer implementiert werden müssen. Im Modell können die Parameter bei der Erstellung der angepassten Elemente beschrieben werden. Für die Umsetzung muss die Sprachdefinition ontologische Erweiterungen zulassen und der Editor eine dynamische Erstellung und Konfiguration von neuen Elementen zulassen.

Konzepte zum Austausch und Anpassung des Modells. Die fünfte der in Abschnitt 2.1 beschriebenen Anforderungen sieht Konzepte vor, wie ein Austausch von Modellelementen und ganzen Modellen erleichtert wird. Durch die Verwendung des XMI zur Serialisierung wird ein Austausch der Modelle vereinfacht. Da alle XMI-Modelle auf den Konzepten der MOF beruhen, können die Modelle untereinander ausgetauscht werden [LLLZ10]. Falls nötig könnte im Editor oder im Code-Generator ein UML-Exporter implementiert werden, der eine Modell-zu-Modell-Transformation ausführt. Falls ein regelmäßiger Austausch des Modelltyps wahrscheinlich wäre, könnte im Code-Generator auch ein Modell-Connector als zusätzliche Abstraktionsebene eingebaut werden, der die Information aus den Modellen ausliest und dann an den Generator bereitstellt. Bei einer Änderung des Modelltyps müsste nur dieser angepasst werden. Durch die Verwendung von Modellierungs-Werkzeugen kann ein Metamodell einer eigenen Sprache als Objektmodell generiert werden, das bei Änderungen automatisch angepasst wird [KH14].

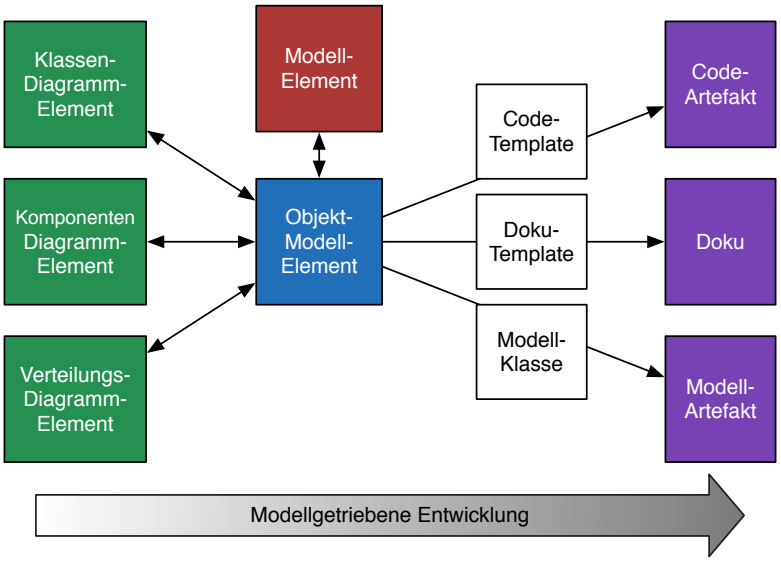


Abbildung 17: Darstellung der möglichen Formen, die ein Modellelement annehmen kann. Der Zugriff auf das Modell geschieht immer über das Java Objektmodell. Der Nutzer bearbeitet das Modell über verschiedene Diagramme, in denen das Modellelement jeweils auf verschieden Arten dargestellt sein kann. Mit dem Element des Objektmodells und einem dazugehörigen Template können dann Projektartefakte generiert werden.

Nutzerdefinierte Abbildung des Modells auf Projektartefakte. Der letzte Punkt den eine modellgetriebene Entwicklungsumgebung definieren muss, sind Konzepte zur Vereinfachung von Nutzer-definierten Abbildungen der Modellelemente zu anderen Projektartefakten, wie dem Quellcode der Software. Wie in Darstellung 17 gezeigt besteht die Beziehung zu beliebigen aus dem Modell generierten Projektinhalten über die EMF-Implementierungen des Metamodells und den Templates. Mit den Templates lassen sich neben Code auch Dokumentationen oder andere Modelle generieren. Über die Anpassung dieser Templates können Nutzer-definierte Änderungen an generierten Artefakten vorgenommen werden. Quellcode wird außerdem mit dem Generation Gap Pattern generiert. Damit kann der Nutzer Änderung an dem generierten Code vornehmen, ohne dass diese beim nächsten Mal überschrieben werden. Über die Möglichkeit der Definition von Basis-Datentypen im Modell lässt sich dieses außerdem leichter an existierenden Code anpassen. Die Definition von eigenen, existierenden Typen wie man sie von dem C “typedef“ kennt können so im Modell verwendet und über die Angabe der einzubindenden Datei auch im generierten Code verwendet werden.

Nachdem die Diagrammtypen aus Abbildung 16 als Modellierungskonzept ausgewählt und die nötigen Modellelemente beschrieben wurden, kann Abbildung 13 erweitert werden. Wie in Abbildung 17 geschieht eine Instanziierung des Modells über ein Java-Objektmodell. Verschiedene Diagramme bieten verschiedene Ansichten auf das Modell, bearbeiten jedoch immer die gleichen Daten. Eine Änderung in einem Diagramm kann auch eine Änderung in einem anderen hervorrufen. Die Modellelemente können also in verschiedenen Diagrammtypen verschiedene Formen annehmen. Die Generierung der Projektelemente aus dem Modell findet über die Kombination der Klassen des Objektmodells mit den gewünschten Templates statt. Die Templates beschreiben also den Zusammenhang der Modellelemente zu den Projektartefakten die daraus generierten werden sollen.

3.3 Verwendete Modellierungssprache

Das Grundkonzept zur Modellierung ist die Erstellung einer an das Tasking Framework angepassten grafischen Sprache, mit der alle Parameter und Eigenschaften des Frameworks beschrieben werden können. Neben der allgemeinen Modellierbarkeit mit der Sprache sollten auch die sonstigen Ziele und Anforderungen umsetzbar sein. So sollte die Sprache zum Beispiel nicht nur alle Parameter beinhalten, sondern auch validiert werden können. Im folgenden Abschnitt wird diskutiert, ob existierende Sprachen verwendet werden können oder eine neue erstellt werden muss.

3.3.1 Verwendung bestehender Sprachen

In Abschnitt 2.2 wurden die drei relevantesten Modellierungssprachen für die Modellierung von eingebetteten Systemen vorgestellt. Alle haben Vorteile und Schwächen jedoch lassen sich die Sprachen auch gemeinsam nutzen. AADL kommt aus dem Bereich der Programmiersprachen und bietet nur geringe Möglichkeiten zur grafischen Modellierung [Niz02]. Über eine Kombination der Sprache mit UML können die Stärken beider Sprachen ausgenutzt werden. Mit UML lassen sich strukturelle Aspekte und die Kommunikation der Software beschreiben während das Laufzeitverhalten mit AADL modelliert werden kann. Neben UML kann auch SysML mit AADl kombiniert werden [DH12]. Da wie in Abschnitt 2.2.2 beschrieben SysML

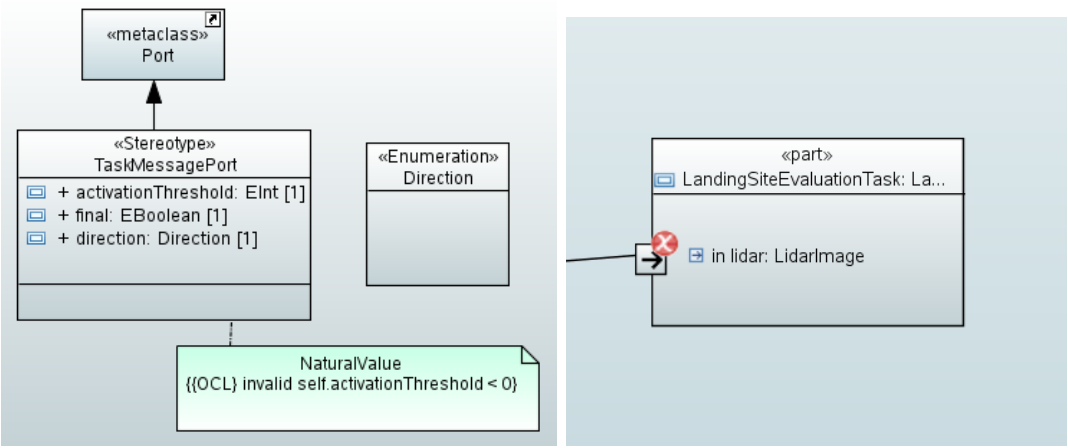


Abbildung 18: Darstellung eines Prototyps zur Erweiterung von UML/SysML-Modellen mit UML-Profilen. In dem links erstellten Profil wird ein Port durch ein Stereotyp erweitert. Der Constraint „NaturalValue“ definiert eine Regel an einen erweiterten Parameter. In der rechten Darstellung zeigt der Editor an, dass die Regel nicht eingehalten wurde.

als UML Profil umgesetzt ist sind diese beiden Sprachen prinzipiell kompatibel zueinander.

Über eine Kombination der Sprachen kann die auf dem Tasking Framework basierende Software modelliert werden. Allerdings wäre eine Kombination aller Sprachen sehr komplex und würde dem Grundziel eines einfachen Editors mit wenigen Fehlerquellen widersprechen. Auch eine Kombination aus UML und SysML könnte mit einem Internen-Block-Diagramm, einem Klassendiagramm und Verteilungsdiagramm alle in 3.2 beschriebenen notwendigen Diagrammtypen umsetzten. Außerdem kann UML über Profile auf ontologischem Weg an das Tasking Framework angepasst werden. Die Verwendung von AADL ist für die grafische Modellierung der zu generierenden Software ungeeignet, da AADL wenige standardisierte Diagramme bietet und das Laufzeitverhalten in den schon existierenden Code-Templates implementiert ist.

In Abbildung 18 ist eine beispielhafte Anpassung einer Kombination eines SysML/UML-Modells dargestellt. Da die links definierte Rahmenbedingung in dem Beispiel fehlschlägt, ist das Element mit dem roten Kreuz versehen. Auf diese Weise ist es vor der Code-Generierung möglich ein Modell zu validieren, um Fehler in diesem zu finden. Undefinierte Zustände im Code-Generator können so verhindert werden.

3.3.2 Erstellung einer eigenen Sprache

Die Erstellung einer grafischen DSL erfordert neben der Beschreibung einer Modellsyntax und -semantik auch eine Definition der Diagramme. Zur Bearbeitung der Diagramme muss zwangsläufig ein Editor implementiert werden. Dieser muss die in dem Diagramm enthaltenen Informationen in ein valides Modell einfügen können. Da es verschiedene Diagrammtypen geben kann, muss der Editor dabei nicht nur wissen, wie die Modellelemente in dem jeweiligen Diagramm dargestellt werden, sondern auch wie eine Bearbeitung des Diagramms das Modell verändert.

Der erste Schritt zur Definition einer eigenen Sprache ist, wie in Abschnitt 2.1.2 beschrieben, die Erstellung eines Metamodells. Mit diesem wird der Aufbau des eigentlichen Nutzermodells beschrieben. Das Modell entspricht dabei einer Datenstruktur, in der die Inhalte der verschiedenen Diagramme konsistent gespeichert werden.

Da die Modellelemente selbst definiert werden, können alle Parameter des Tasking Frameworks im Modell verwendet werden. Bedingt durch die Reduzierung von nicht benötigten Parametern aus Sprachen wie UML, wird das Modell so deutlich einfacher. Die grafische Darstellung des Modells in Diagrammen, eine syntaktische Überprüfung und anschließende Validierung müssen jedoch selbst implementiert werden. Im Gegensatz zu einer Lösung mit einer existierenden Sprache können Syntax und Semantik der Diagramme dabei so konkret definiert werden, dass nur für das Tasking Framework gültige Elementkombinationen erstellt werden können.

Der Vorteil einer eigenen Sprache ist also die hohe Flexibilität, das einfache Nutzermodell und die mögliche, eigene Syntaxüberprüfung. Der Nachteil ist der höhere Aufwand durch die Erstellung des Metamodells, der Diagramme und des Editors.

3.3.3 Auswertung und Entscheidung

Abschnitt 3.3.1 beschreibt, dass eine Software die auf dem Tasking Framework beruht mit allen, für die Code-Generierung notwendigen Aspekten und Parametern beschrieben werden kann. Da es schon fertige Editoren für UML und SysML gibt, kann im Vergleich zu einem Ansatz, bei dem eine neue Sprache definiert wird, sehr viel Arbeit eingespart werden. Eine Umsetzung mit Papyrus ermöglicht die Verwendung von UML als grafischer DSL, die für modellgetriebene Code-Generierung verwendet werden kann.

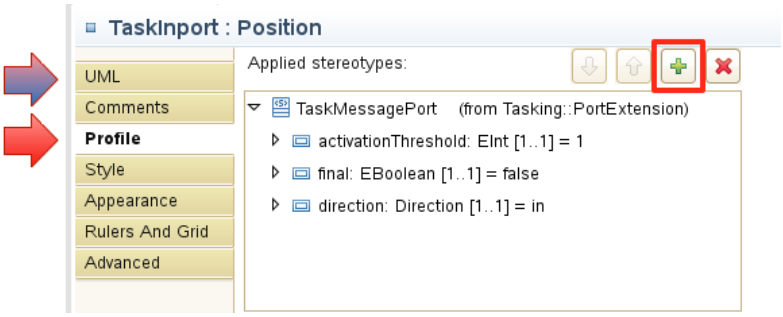


Abbildung 19: Screenshot des UML-Editors Papyrus. Um die erweiterten Attribute eines Elements zu verwenden muss auf den Properties Tab gewechselt, der Stereotyp angewendet werden. Alternativ können Stereotypen auch in die Diagramm-Palette eingefügt werden.

Da bei der Definition einer eigenen Sprache alle notwendigen Konzepte selbst umgesetzt werden können, ist die modellgetriebene Entwicklung auch mit einem solchen Ansatz möglich.

Da beide Konzepte umsetzbar sind, berücksichtigt die Entscheidung neben dem Vergleich mit den Anforderungen auch wie aufwändig die jeweiligen Umsetzungen und spätere Verwendung der Modellierung sind.

Die Studie „A systematic review on the definition of UML profiles“ beschreibt einen Rückgang der Verwendung von UML-Profilen seit ca. 2006 und sieht mögliche Gründe dafür bei der hohen Komplexität und dem Aufkommen von Werkzeugen wie dem GMF, die eine Erstellung eigener Sprachen ermöglichen [Par10]. Dabei ist weniger die Erstellung eines Profils die Herausforderung, als die Modellierung mit diesem. So muss ein Modell, welches ein Profil verwenden soll, dieses immer erst manuell aktivieren. Die erweiterten Attribute eines Modellelements sind zudem nicht unter den normalen UML-Parametern zu finden, sondern wie in Abbildung 19 dargestellt, bei allen gängigen EMF-UML-Editoren in einem zusätzlichen UML-Profil-Beriech. Für die Verwendung von, durch Profilen erweiterten, Modellen zur modellgetriebenen Softwareentwicklung muss der modellierende Entwickler also noch zusätzliches Wissen über die Verwendung von Profilen besitzen.

Da die Syntax einer Sprache in den Diagrammen und somit deren Editoren implementiert ist, wird eine nachträgliche, grundlegende Anpassung der Syntax nur über Anpassungen der Diagrammtypen und Editoren möglich. Eine Einschränkung von UML über semantische Constraints ermöglicht zwar ein Validierung der Eingaben in Bezug auf das Tasking Framework,

allerdings ist es nicht möglich, dass der Editor für das Tasking Framework ungültige Konstrukte gar nicht erst zulässt. Der Editor lässt weiterhin alle für UML gültigen Kombinationen der Elemente zu, auch wenn diese über UML Constraints eingeschränkt wurden. Die Validierung weist den Nutzer dann darauf hin, dass das Modell nicht valide ist und beschreibt die Fehler. Der modellierende Entwickler muss das Modell daraufhin korrigieren.

Abschnitt 2.1.2 beschreibt, dass eine geringe Beschreibung der Syntax hohe Anforderungen an die Semantik einer Sprache stellt. Für eine Umsetzung des Tasking Frameworks über Constraints in einem Profil, die nur über eine semantische Validierung im Editor umgesetzt ist, bedeutet dies, dass die Beschreibung aller notwendigen Constraints sehr aufwendig ist. Da die Sprache UML sehr mächtig ist, können außerdem nur sehr schwer alle Möglichkeiten einer Fehleingabe ausgeschlossen werden. Die Constraints müssten bei einer neuen Version von UML angepasst werden und durch die hohe Anzahl an notwendigen Constraints besteht zusätzlich eine hohe Fehleranfälligkeit des Profils.

Mit einer Kombination und Erweiterung bestehender Sprachen ist es möglich, die Komplexität des Tasking Frameworks vollständig zu beschreiben. Dies bedeutet allerdings sowohl für die erstellten Erweiterungen als auch für die späteren Arbeiten am Modell einen hohen Aufwand. Mit bestehenden Editoren ist es nur über nachträgliche Validierung möglich, den modellierenden Entwickler auf fehlerhafte Eingaben hinzuweisen. Ein vollständiger Ausschluss aller möglichen Falscheingaben ist zudem kaum möglich. Aufgrund der Anforderung, dass nur gültige Konstrukte zugelassen werden sollen und die Verwendung des Editors möglichst einfach sein soll, wurde entschieden, dass eine neue Sprache zur Modellierung des Tasking Frameworks sinnvoll ist.

3.4 Sprachdefinition

Der erste Schritt zur Definition einer Sprache ist, wie in Abschnitt 2.1.2 beschrieben, die Erstellung eines Metamodells. Dieses Metamodell beschreibt die Syntax und Semantik des Nutzermodells. Zur grafischen Modellierung der Software müssen zusätzlich auch noch Diagramme definiert werden. Diagramme können unterschiedliche Bereiche des Modells auf unterschiedliche Arten darstellen. Nicht alle Modellelemente müssen in jedem Diagramm zu sehen sein. Da die Diagramme ein valides Modell erstellen müssen, macht die Verwendung der Modellsyntax in den Diagrammen Sinn.

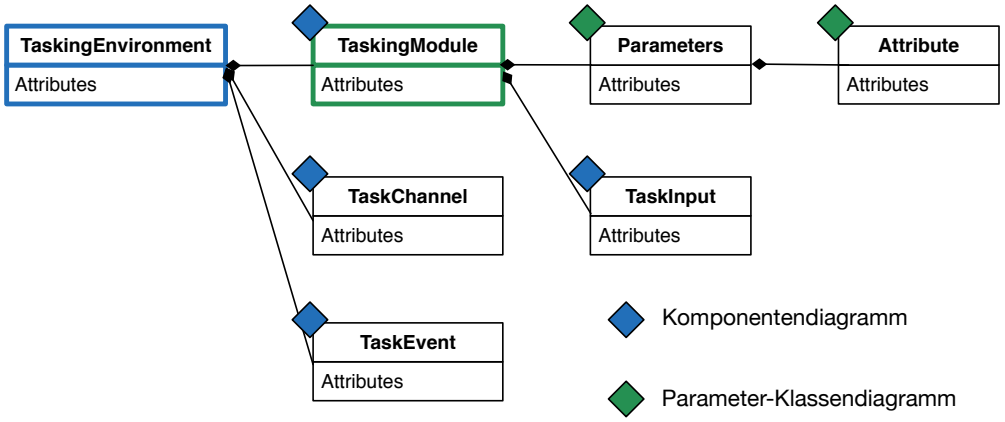


Abbildung 20: Konzeptionelle Darstellung zur Beschreibung eines Ausschnitts der konkreten Syntax. Zur Vereinfachung wurden abstrakte Modellelemente nicht dargestellt. Die Quadrate geben jeweils an in welchem Diagramm die Modellelemente dargestellt werden. Das Wurzelelement eines Diagramms wurde jeweils mit der Diagrammfarbe markiert.

Neben den eigentlichen Modellelementen die über das Konzept der Klassen im Metamodell beschrieben werden, sollten auch abstrakte Modellelemente definiert werden. Um nicht jedem Modellelement einen Namen geben zu müssen können alle Modellelemente von einem „Named-Element“ erben. Diese abstrakten Elemente ermöglichen es auch mehrere Modellelemente mit einem Constraint zu validieren. Eine Überprüfung, ob der Name eines Elements gesetzt ist, lässt sich so generisch für alle Elemente beschreiben. In Abbildung 20 ist eine erste konzeptionelle Darstellung der Beschreibung eines Teils der Syntax des Modells mit den Elementen des Tasking Frameworks dargestellt. In das Wurzelelement des Modells dürfen jeweils Software-Module, TaskChannels und TaskEvents eingefügt werden. Parameterklassen und TaskInputs dürfen jeweils nur Modulen untergeordnet werden.

Die einfachste Methode Diagramme zu definieren ist die, ein Wurzelelement eines Diagramms anzugeben und für die Kinder dieses Elements zu beschreiben, welche Modellelemente in dem Diagramm dargestellt werden. Auf diese Weise können Syntax und Semantik des Modells für die Diagramme übernommen werden. Wenn ein Element in dem Modell nicht als Kind des Wurzelelements eingefügt werden kann, darf der Editor das im Diagramm auch nicht zulassen. In Abbildung 20 sind Teile der Diagrammdefinition für das Komponentendiagramm und zur Beschreibung der Parameter eines Moduls eingefügt. Das Komponentendiagramm wird das wichtigste Diagramm, da mit diesem die Softwaremodule und deren Kommunikationswege

definiert werden. Da dieses Diagramm die Grundstruktur der Software beschreibt, muss das Wurzelelement des Diagramms dem Wurzelelement des Modells entsprechen. Die Parameter eines Softwaremoduls sollen dagegen nicht in dem Komponenten- sondern in einem Klassendiagramm dargestellt werden. Das Wurzelelement eines solchen Klassendiagramms zur Beschreibung der Parameter eines Moduls ist aus diesem Grund das Softwaremodul selbst. Zur Erstellung eines solchen Diagramms muss deshalb angegeben werden, welches Modul das Diagramm beschreiben soll. In das Diagramm können dann Parameterklassen mit den Parametern als Attributen eingefügt werden. TaskInputs werden nur in dem Komponentendiagramm dargestellt, da diese zur Konfiguration der Ereignisstuerung des Komponentendiagramms gehören.

Neben der Beschreibung welche Modellelemente in einem Diagramm dargestellt werden, muss auch definierten werden, wie die Elemente in dem Diagramm jeweils aussehen sollen. Da die Modellelemente je nach Diagrammtyp unterschiedliche Formen annehmen können, reicht eine grafische Notation pro Modellelement nicht mehr aus.

4 Erweiterung des Tasking Framework mit modellgetriebenen Entwicklungskonzepten

Im folgenden Kapitel ist die Umsetzung der modellgetriebenen Entwicklung mittels einer eigenen Sprache beschrieben. Der Erste Schritt ist dabei die Auswahl der Werkzeuge zur Definition und Implementierung der Modellierungssprache und Diagramme. Nach der Auswahl der Werkzeuge folgt die Erstellung des Metamodells und anschließend die Implementierung der Diagrammeditoren.

4.1 Werkzeuge zur Erstellung von Modelleditoren

In Abschnitt 1.2.3.4 wurde beschrieben, dass ein Werkzeug zur modellgetriebenen Entwicklung auf dem Eclipse Modeling Framework (EMF) beruhen soll, da der existierende Code-Generator mit diesem Framework umgesetzt ist. Das EMF ist ein quelloffenes Werkzeug zur Modellierung von beliebigen Anwendungsgebieten. Im Zentrum des Frameworks steht die Sprache Ecore, die auf dem XMI basiert und eine Sprache zur Erstellung von Metamodellen ist [KH14]. Zur Erstellung eines Metamodells werden mehrere grafische und textuelle Editoren bereitgestellt. Für ein Metamodell generiert das EMF eine Java Implementierung, mit der wie in Abschnitt 2.1.3 beschrieben, Instanzen der im Metamodell definierten Sprache angelegt und serialisiert werden können.

Da der Code-Generator auf dem EMF beruht, macht es Sinn, das Metamodell in der Sprache Ecore zu erstellen. Aufbauend auf einem solchen Ecore-Metamodell gibt es verschiedene Frameworks zur Erstellung von Diagrammen. Die bedeutendsten in Eclipse aufgenommenen Projekte zur Implementierung von Diagrammeditoren sind das Graphical Modeling Framework (GMF) und Graphiti. Beide ermöglichen die Definition von Diagrammelementen und stellen eine Basis-Implementierung eines Editors bereit. Die Werkzeuge haben jedoch unterschiedliche Philosophien. Während das GMF den Fokus auf die optimale Darstellung des Modells legt, versucht Graphiti dem User eine vom EMF unabhängige Java-API zu bieten, mit der die

optischen Aspekte der Diagramme besonders leicht bearbeitet werden können [BGK+10]. Da die Wahl des Werkzeugs weitreichende Folgen für Modellierung hat, lag großer Wert auf dem Vergleich beider Möglichkeiten Diagrammeditoren zu erstellen.

In der Masterarbeit von Ivar Refsdal [Ref11] wurden die beiden Werkzeuge unter anderem unter dem Gesichtspunkt der Aufwandsabschätzung zur Erstellung von Diagrammeditoren verglichen. Da diese Arbeit jedoch nicht das Ziel einer modellgetriebenen Entwicklung hat, können die Ergebnisse der Arbeit nicht direkt übernommen werden. Eine Anwendung für modellgetriebenen Entwicklung legt mehr Wert auf die funktionalen also auf die optischen Aspekte, da die Diagramme Mittel zum Zweck sind. Das Ergebnis der Arbeit sieht einen Vorteil bei Graphiti, da der generierte Code deutlich geringer und übersichtlicher ist. Zudem sind die generierten Diagramme leichter anpassbar. Der Autor sieht allerdings Vorteile bei einer Verwendung des GMFs je komplexer das Metamodell ist. Graphiti speichert die Daten standardmäßig zudem nicht in einem Modell, sondern in getrennten Diagramm-Dateien. Eine Verwaltung des Modells muss so vom Nutzer implementiert werden. Ein weiterer Vorteil des GMFs ist, dass generische Validierung verwendet werden kann. Das GMF implementiert die Eclipse Fehlerdarstellung und zeigt so Probleme im Editor analog zu z.B. Syntaxfehlern bei textuellen Sprachen an. Graphiti verwendet sowohl für die Darstellung als auch für die Beschreibung der Rahmenbedingungen nur eigene Lösungen und bietet somit sehr viel weniger Flexibilität.

Da sich eine Verwendung der Diagrammeditoren zur modellgetriebenen Entwicklung anbietet gibt es sowohl für Graphiti als auch für das GMF Generatoren, die große Teile der jeweiligen API generieren. Die Entwicklung mit Graphiti lässt sich durch ein Werkzeug mit dem Namen Spray effektiv verkürzen [FKBG]. Dabei kann die grafischen Darstellung der Diagrammelemente stark angepasst werden. Auch wenn das Paper die deutliche Zeitersparnis durch ein Verwendung von Spray hervorhebt liegt der Fokus dennoch auf den grafischen Möglichkeiten der Diagramme. Im Gegensatz dazu gibt es mit EuGENia ein Werkzeug, das auf dem GMF aufbaut und einen effektiven Weg zur Definition von Diagrammen aufzeigt [DALR15]. Ähnlich dem in Abschnitt 3.4 beschriebenen Konzept zur Beschreibung von Diagrammen lassen sich Diagramme durch Annotationen im Metamodell definieren. Diese Annotationen entsprechen den farblichen Markierungen in Abbildung 20 und ermöglichen die Generierung eines einfachen Editors. Ein so generierter Diagrammeditor setzt bereits große Teile der Konzepte von GMF-Editoren um und ermöglicht die Erstellung eines Diagramms, in dem nur valide Modellkonstrukte zugelassen, eine generische Validierung ermöglicht und die Nutzerdaten in

einem Modell speichert. Der einzige Nachteil dieses Werkzeugs ist, dass es nur ein Diagramm pro Modell generieren kann. Da wie in Abschnitt 3.2 beschrieben jedoch mehrere Diagramme notwendig sind, reicht eine Verwendung dieses Werkzeugs zur Erstellung der Diagramme nicht aus.

EuGENia ist ein Teil des Epsilon-Projekts. Epsilon steht für „Extensible Platform of Integrated Languages for mOdel maNagement“ und stellt mehrere Werkzeuge und Sprachen zur Entwicklung mit Modellen bereit [Dim15]. Teil davon sind Sprachen zur Modell-zu-Modell-Transformation, zum Mergen von Modellen und zur Beschreibung von Constraints.

4.2 Bewertung der Technologien

Auf Grund der großen Tragweite der Auswahl des Werkzeugs zur Erstellung des Diagrammeditors wurde entschieden, dass eine Prototyp-Implementierung beider Möglichkeiten sinnvoll ist.

4.2.1 Prototypen

Graphiti mit Spray Zur Erstellung eines Prototyps mit Spray wurde nach der offiziellen Anleitung vorgegangen [WTR]. Bereits an der API von Spray wird deutlich, dass der optischen Darstellung der Diagramme besondere Bedeutung obliegt. So gibt es eine eigene textuelle DSL zur Beschreibung der Darstellung der Diagrammelemente. Spray generiert Graphiti-Code und ermöglicht Anpassungen des generierten Codes über das Generation-Gap Pattern. Die Graphiti API besteht aus vom EMF unabhängigen Java-Klassen. In Abbildung 21 ist ein erster Editor dargestellt. Standardmäßig bietet der Editor dem Nutzer schon sehr viele Features. Bei der Positionierung der Elemente werden Hilfslinien angezeigt und der Editor bietet ein Raster, an dem die Elemente angeordnet werden können. Trotz dieser schon standardmäßig vorhandener Funktionen müssen insbesondere für eine modellgetriebene Entwicklung noch viele Anpassungen an dem Editor vorgenommen werden. Neben der Verwaltung des Modells muss auch implementiert werden, wie die Diagramme zusammenarbeiten. Eine Änderung in einem Diagramm führt nicht automatisch zu einer Aktualisierung der anderen.

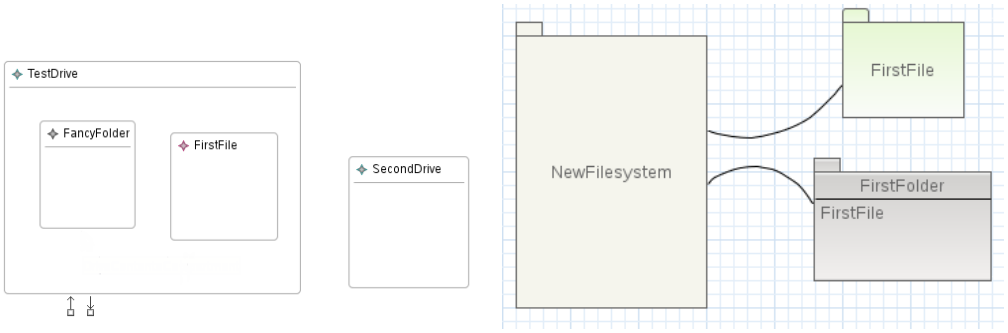


Abbildung 21: Screenshots der Prototypen eines mit EuGENia (links) und Spray (rechts) erstellten Editors. Das mit Spray generierte Diagramm liefert optisch schönere Ergebnisse. Standardmäßig bietet der Spray-Editor auch Hilfslienien und ein Raster mit dem die Elemente angeordnet werden können.

Graphical Modelling Framework mit EuGENia Zur Entwicklung von Editoren mit EuGENia kann eine fertig konfigurierte Eclipse-Entwicklungsumgebung von der Epsilon-Projekt-Seite geladen werden. Diese enthält alle Werkzeuge zur Modellierung und Erstellung von Editoren mit dem GMF. Nach der Definition eines Metamodells kann ein Diagramm über Annotation im Modell definiert werden. Die Annotationen beschreiben welche Modellelemente in dem Diagramm dargestellt werden sollen. Ein Modellelement kann als Verbindung oder als Knoten dargestellt werden. Neben der Auswahl der darzustellenden Objekte muss auch das Wurzelelement des Diagramms definiert werden. Aus dem so angepassten Metamodell lassen sich komplexe Diagramme mit geringem Aufwand generieren. EuGENia generiert GMF-Modelle, die dann zur Erstellung des Diagramm-Codes verwendet werden. Anpassungen können dabei nachträglich sowohl an den GMF-Modellen als auch am generierten Code vorgenommen werden. Im generierten Code kann für jede Funktion oder Klasse getrennt beschrieben werden, ob das Elemente bei der nächsten Generierung überschrieben werden soll oder nicht.

Mit dem GMF müssen die Diagramm in Modellen beschrieben werden. Dabei gibt es ein Modell zur Beschreibung der Darstellung der Modellelemente, ein Modell zur Beschreibung der Verbindung von Modell- und Diagrammelementen und ein Generator-Modell mit dem Pfade, Ressourcen und Quellcode-Namen für den zu generierenden Code beschrieben werden. Aus diesen Modellen kann dann der Diagrammcode generiert werden. EuGENia bietet eine Möglichkeit mit minimalem Aufwand die sehr komplexen GMF-Modelle zu erstellen. Über die Annotation im Metamodell können die Grundlagen der optischen Darstellungen beschrieben werden. Zur weiteren Anpassung müssen die GMF-Modelle und der daraus generierte Dia-

grammcode angepasst werden. Der GMF-Code ist sehr viel größer, komplexer und am Anfang unübersichtlicher als der Code von Graphiti. Nahezu im gesamten Diagrammcode wird das EMF und darauf aufbauende Frameworks verwendet. Für alle Änderungen im Diagramm gibt es Kommandos, in denen beschrieben wird, wie das Modell angepasst werden muss. Änderungen am Modell werden immer über Transaktionen vorgenommen. Nur wenn eine Transaktion vollständig ausgeführt wurde, werden die Änderungen im Modell übernommen. Über Transaktionen wird auch zugesichert, dass der Zustand des Modells konsistent ist, da immer nur eine Transaktion gleichzeitig ausgeführt wird.

Mit EuGENia kann nur ein Diagramm pro Modell erstellt werden, da das Werkzeug jedoch GMF Code generiert, ist eine Möglichkeit zur Erweiterung des Editor die Implementierung der anderen Diagrammtypen direkt mit den Mittel des GMF. Nach einigen Versuchen war es möglich ein zweites, mit GMF erstelltes Diagramm, in den Editor zu integrieren.

4.2.2 Auswertung und Auswahl

Die Erstellung der Prototypen hat die bei der Literatur-Recherche bereits vermutete Aufteilung der Anwendungsgebiete bestätigt. Die Philosophie von Graphiti ist die möglichst einfache Erstellung eines Diagrammeditors. Dafür wird die komplexe EMF-API abgekapselt und eine eigene, deutlich einfachere API bereitgestellt. Durch die Eigenimplementierung vieler Elemente des Editors wird dieser deutlich einfacher, allerdings werden so auch viele Möglichkeiten des GMF ausgeschlossen. Durch eine eigene Validierung ist es nicht mehr möglich die generischen EMF-Constraints zu verwenden und die Fehler werden nicht in im Eclipse-Problembereich angezeigt. Auch der Algorithmus zur automatischen Anordnung der Elemente des GMFs kann nicht mehr verwendet werden.

Da der Editor zur modellgetriebenen Entwicklung eingesetzt werden soll und dabei das Modell im Zentrum steht, macht nur die Verwendung des GMFs Sinn. Graphiti bietet zwar optisch einen anspruchsvolleren Editor, allerdings wird man so von der Graphiti-API abhängig. Schon die fehlende Möglichkeit der generischen EMF-Validierung zeigt, dass durch die eigene API Funktionen verloren gehen. Da für die modellgetriebene Entwicklung die Optik zweitrangig ist, macht es Sinn einen Editor mit der generischen GMF-API zu erstellen, auch wenn diese etwas aufwendiger ist.

4.3 Implementierung der Entwicklungsumgebung

Die eigentliche Implementierung erfolgte nun mit den ausgewählten Werkzeugen. Das Metamodell wird mit der Metasprache Ecore des EMFs erstellt, die Diagramme werden mit EuGENia und dem GMF definiert. Mit dem Diagrammeditor wird ein wichtiges Werkzeug zur modellgetriebenen Entwicklung bereitgestellt. Nur wenn die Verwendung des Editors einfach ist, wird die modellgetriebene Entwicklung langfristig verwendet. Neben der Nutzbarkeit des Editors sollte dieser auch alle im Diagramm enthaltenen Informationen ins Modell übernehmen. Eine im Diagramm dargestellte Relation, die nicht vollständig im Modell enthalten ist kann in der Code-Generierung eventuell nicht verwendet werden.

4.3.1 Erstellung des Metamodells

Die Definition des Metamodells erfolgt mit Sprache Ecore. Das EMF bietet dafür einen Editor, in dem in einem Klassendiagramm ähnlichen Diagramm, die Elemente beschrieben werden können. Abbildung 22 zeigt ein Diagramm des Metamodells ohne abstrakte Elemente, das vollständige Metamodell ist in Anhang A dargestellt. Von Epsilon gibt es eine Sprache die eine textuelle Darstellung des Ecore-Modells liefert [Eclb]. Mit dieser können die Modellelemente in einer Java-ähnlichen Syntax beschrieben werden. Mit dem EMF wird eine Java-Implementierung des Metamodells generiert, über die der Editor und der Code-Generator dann auf das Modell zugreifen können.

Abstrakte Typen Neben den vom Tasking Framework benötigten Modellelementen gibt es auch abstrakte Elemente. Das Metamodell zur Definition der Tasking Language hat neben Elementen wie Typed- / NamedElement z.B. auch ein Classifier-Element. Classifier können von anderen erben und enthalten Attribute. Im Modell sind z.B. Klassen, Datentypen und Stereotypen Classifier. Von abstrakten Elementen können nicht nur Attribute geerbt werden, sondern in die Metamodell-Java-Implementierung kann auch Logik eingebaut werden. Im Classifier wurde die Getter-Methode z.B. so implementiert, dass auch die Attribute von den Superelementen zurückgegeben werden.

Datentypen Im Metamodell muss auch die Verwaltung der Datentypen geregelt sein. Die Java / Ecore Datentypen können nicht einfach auf die Datentypen der Zielsprache des

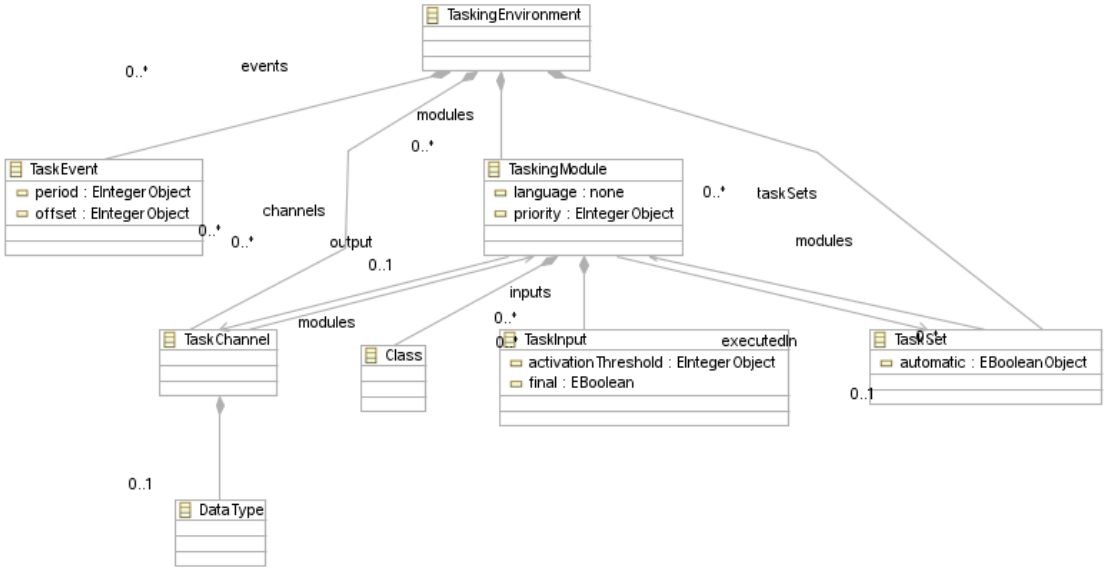


Abbildung 22: Grafische Darstellung eines Metamodells in der Ecore-Sprache. Zur Vereinfachung sind auch hier abstrakte Elemente nicht dargestellt. Die Kompositionen geben an, dass ein Element ein anderes im Nutzermodell enthalten kann.

generierten Codes übertragen werden. So gibt es in C z.B. sehr viele verschiedene Integer-Typen mit verschiedenen Wortbreiten. Da in Sprachen wie C auch Typen umdefiniert und in Ada grundsätzlich eigene Datentypen definiert werden, macht es keinen Sinn alle möglichen Datentypen im Modell anzubieten. Aus diesem Grund wurde entschieden, dass im Modell auch eigene Basistypen definiert werden können. Neben dem Namen besitzt ein Basistyp auch Attribute zur Beschreibung, welche Ressourcen für den Typ eingebunden werden müssen.

In eingebetteten Systemen der Sprache C enthalten die Zeiger auf ein Array keine Informationen über die Länge des Arrays. Dieses unter dem Namen „Pointer-Degeneration“ bekannte Problem hat zur Folge, dass Arrays nicht einfach als Zeiger übergeben werden können. Um dennoch Arrays in der generierten Software verwenden zu können, werden diese in Klassen abgekapselt, die jeweils für eine Dimension eine Variable mit der Länge besitzen. Auf die Daten werden dann mit generierten Getter und Setter Methoden zugegriffen. Im Modell können Arrays definiert werden, indem Dimensionen und deren Größe angegeben werden. Auf diese Weise ist es möglich auch nicht symmetrische, mehrdimensionale Arrays zu erstellen.

Während TaskChannels nur einen Datentyp als Typ besitzen, können Attribute Basisdaten-

```
1 @gmf.node(label = "name", figure="rectangle", size="100,50")
2 class TaskingModule extends NodeElement{
3  attr Integer priority = 100;
4
5  @gmf.link(target.decoration="arrow")
6  ref TaskSet executedIn;
7  [...]
8 }
```

Quellcode 1: Ausschnitt aus der textuellen Darstellung eines Ecore-Modells in der Emfatic-Sprache. Über Annotationen werden die grafische Darstellungen im Diagramm beschrieben. Das TaskingModule wird als Rechteck mit der Größe (100,50) dargestellt. Die Referenz auf das TaskSet wird im Diagramm als Verbindung in Form eines Pfeils dargestellt.

typen, Arrays, modellierte Klassen und Datentypen als Typ enthalten. Auch hier wird ein abstraktes Elemente „Type“ verwendet, den alle diese Elemente implementieren.

Zur Umsetzung von defensiver Programmierung, wurde das Prinzip der „Ranges“ in das Modell aufgenommen. Für Attribute können obere und untere Grenzen angegeben werden, zwischen denen der Wert des Attribut sich befinden muss.

4.3.2 Erstellung der Diagrammeditoren mit EuGENia

Zur Definition der Diagramme wird die textuelle Darstellung des Ecore-Metamodells verwendet. Über Annotationen werden die Elemente, die in einem Diagramm dargestellt werden sollen markiert. Mit den Annotationen kann wie in Quellcode 1 dargestellt, beschrieben werden, ob ein Modellelement ein Knoten oder eine Verbindung zwischen Knoten ist und mit welcher Form das Element dargestellt werden soll. Mit dieser Erweiterung des Metamodells kann EuGENia nun GMF-Modelle zur Beschreibung des Editors generieren. Der nächste Schritt zur Erstellung eines Diagrammeditors ist die Code-Generierung aus den GMF-Modellen.

Zur Anpassung des generierten Editors können sowohl die GMF-Modelle als auch der generierte Diagrammcode bearbeitet werden. Optische Änderungen an den Diagrammelementen lassen sich gut in den Modellen bearbeiten, tiefer gehende Änderungen können besser im Diagrammcode vorgenommen werden.

Eine essentielle Funktion der modellgetriebenen Entwicklungsumgebung ist die Möglich-

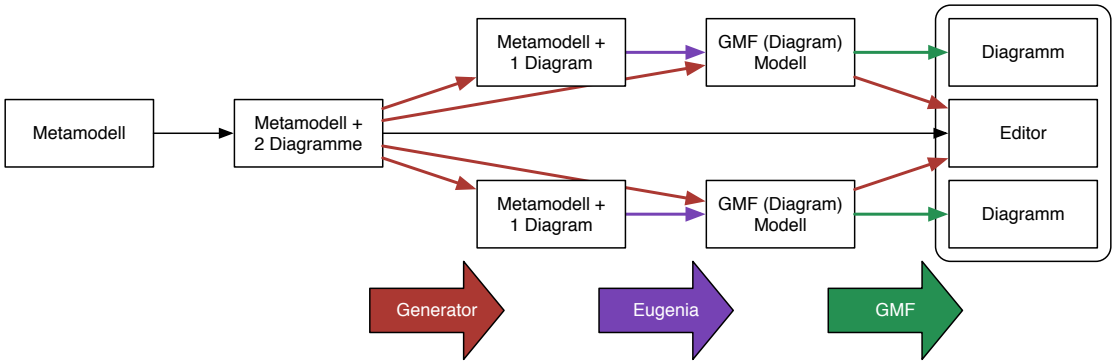


Abbildung 23: Darstellung des Vorgehens zur Erstellung eines Diagramm-Editors. Da EuGENia nur die GMF-Modelle eines Diagramms pro Modell ermöglicht, wurde ein Generator erstellt, der dieses aufteilt. Zusätzlich werden EOL-Skripte generiert, die die GMF Modelle so anpassen, dass die Diagramme in einem gemeinssamen Editor verwendet werden können.

keit mehrere Diagrammtypen zur Erstellung des Modells verwenden zu können. Über eine Anpassung am Editor unterstützt das GMF die Erstellung mehrerer Diagramme zu einem Metamodell [Eclc]. Dabei ist es auch möglich, die Editoren so zu konfigurieren, dass die Diagramme untereinander automatisch aktualisiert werden.

Nachdem es möglich war, mit GMF-Mitteln mehrere Diagramme zu erstellen, war der nächste Entwicklungsschritt die Analyse der notwendigen Änderungen, um den EuGENia-Workflow so anzupassen, dass weitere Diagramme auch mit EuGENia definiert werden können. Da mit der Syntax des Frameworks nur ein Diagramm pro Metamodell definiert werden kann, war der erste Schritt die Erweiterung der Syntax zur Defintion mehrere Diagrammelemente. Die einfachste in der Ecore-Sprache validen Syntax ist, die Erweiterung der Annotation um die Angabe des Diagramms. Aus „@gmf.node(...)“ wird so „@gmf.diagramType.node(...)“. Um EuGENia weiterhin verwenden zu können, muss das Metamodell mit mehreren Diagrammen in ein Metamodell mit nur einem Diagramm aufgeteilt und die Annotationen angepasst werden. Nachdem für die separaten Diagrammdefinitionen die GMF-Modelle generiert wurden, müssen diese so angepasst werden, dass es möglich ist, alle Diagrammeditoren in einer Eclipse-Instanz zu verwenden. Um diesen Arbeitsablauf zu automatisieren, wurde, wie in Abbildung 23 dargestellt, ein Generator implementiert, der die neue Syntax zur Definition mehrerer Diagramme parsen und die Diagramme aufteilen kann. Der Generator erkennt dabei alle im Metamodell enthaltenen Diagramme automatisch und erstellt für diese Unterordner

mit der zu EuGENia kompatiblen Diagrammdefinition. Zusätzlich werden noch Skripte in der Sprache EOL generiert, die die GMF-Modelle so anpassen, dass die Diagramme parallel verwendet werden können und sich gegenseitig aktualisieren. EOL steht für „Epsilon Object Language“ und ist eine Sprache des Epsilon-Projekts, die eine Modell-zu-Modell-Transformation ermöglicht. Beim Generieren der GMF-Modelle mit EuGENia werden die Skripte automatisch ausgeführt. Zur Erstellung eines neuen Diagramms müssen nur die neuen Notationen in das Metamodell eingefügt, der Generator ausgeführt und der Diagrammcode mit EuGENia generiert werden. EuGENia startet dabei selbständig die Code-Generierung aus den GMF-Modellen.

4.3.3 Grafische Anpassung der Editoren

Mit EuGENia lassen sich unter geringem Aufwand Diagramme erstellen, bei denen für die Diagrammelemente eine Form und eine Größe angeben werden kann. Zur Verwendung von eigenen Figuren, Icons und Texten im Diagramm muss der Diagramm-Code angepasst werden. Spezielle Layouts und optische Elemente können auch im GMF-Modell eingefügt werden.

Um die Diagramme möglichst verständlich zu gestalten, wurde z.B. das Klassendiagramm zur Modellierung der Parameter und Datentypen an das UML-Design angepasst (siehe Abbildung 24). Dazu musste das Layout des Klassen-Elements verändert, ein Linie unter den Namen eingefügt und der Rahmen des Attributs entfernt werden. Um auch den Namen des Typs im Titel anzuzeigen musste der Diagrammcode angepasst werden. Standardmäßig können im Titel des Elements nur eigene Elemente des Typs String stehen. Um den Typ dennoch anzuzeigen, musste ein eigener Attributparser geschrieben werden, der auch Referenzen des Elements darstellen und deren Namen anzeigen kann.

Neben der grafischen Anpassung an ganzen Diagrammen wurden auch einzelne Diagrammelemente angepasst. Da die Ereignissteuerung des Tasking Framework, wie in Abschnitt 3.1.2 beschrieben, hauptsächlich auf dem Senden von Signalen durch TaskChannels und dem Empfangen durch TaskInputs beruht, orientieren sich die Diagrammdarstellung beider Elemente an den UML-Elementen von Signal-Sender und Empfänger. Wie in Abbildung 25 dargestellt enthält das Diagrammelement der TaskChannels zusätzlich noch eine schematische Darstellung der Art des enthaltenen Datenspeichers. Auf diese Weise kann beim Betrachten des Komponentendiagramms direkt erkannt werden welche Datenspeicher an welchen Stellen

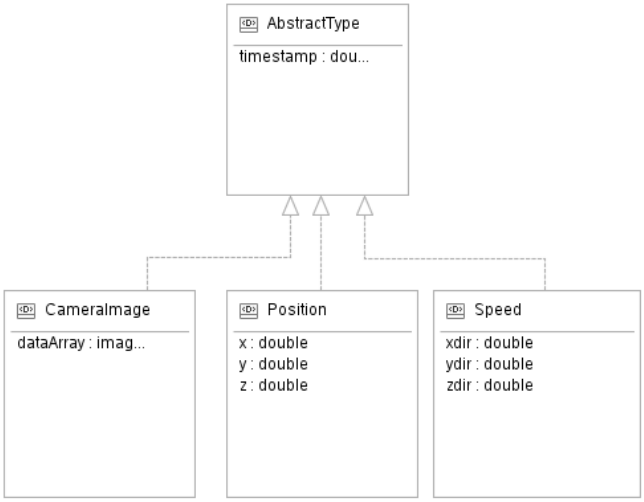


Abbildung 24: Screenshot des Editors eines Klassendiagramms. Das Diagramm wurde so angepasst, dass es einem UML-Klassendiagramm möglichst ähnlich ist. Neben den nicht gefüllten Pfeilen stehen die Klassenattribute als Textelemente mit Name und Typ im Klassenkörper.

im System verwendet wurden.

Weitere Bedeutung in den grafischen Notationen stellen die Ecken da. Elemente wie TaskEvent, TaskInput oder TaskSet die vom Tasking Framework selbst implementiert sind und eine spezielle Bedeutung für das Framework haben, werden in der Form eines Rechtecks dargestellt, während Software-Module, die jegliche Art von Software enthalten können abgerundete Ecken besitzen.

4.3.4 Anpassung des Zusammenhangs zwischen Modell und Diagramm

Für jedes Modellelement gibt es eine tabellarische Ansicht, in der alle Eigenschaften des Modellelements und deren Inhalt aufgeführt sind. Diese Ansicht wird angezeigt sobald ein Element im Diagramm ausgewählt wurde. Die Eigenschaften eines Modellelements können entweder direkt durch den Nutzer über diese tabellarische Ansicht oder durch die Diagramme gesetzt werden. Wird im Diagramm eine Verbindung von einem TaskChannel zu einem TaskInput gezogen, wird in der Liste der Eigenschaften des TaskChannels der neue TaskInput automatisch als neuer Empfänger hinzugefügt. Eigenschaften wie die Priorität eines Tasks

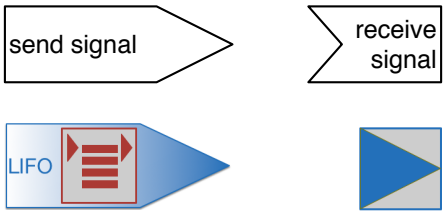


Abbildung 25: Darstellung der Elemente des Benachrichtigungssystems des Tasking Frameworks. Die TaskChannels sind eine Kombination des UML-Elements eines Signal-Senders und einer Darstellung der Art des beinhalteten Datenspeichers, während sich das Diagrammelement des TaskInput am Signal-Empfänger orientiert.

die nicht in Diagrammen beschrieben werden, müssen vom Nutzer in dieser Ansicht gesetzt werden. Um zu verhindern, dass Referenzen, die durch das Diagramme erstellt wurden, manuell über die Ansicht der Eigenschaften zerstört werden, wurde die Ansicht so implementiert, dass diese dort nicht bearbeitet, sondern nur angesehen werden können.

Wie in Abschnitt 4.2.1 beschrieben ist der GMF-Diagrammcode sehr stark Modell-orientiert. Im Diagrammeditor gibt es für jedes Element eine eigene Kommandoklasse in der alle Änderungen am Modell vorgenommen werden, die notwendig sind, sobald ein Element im Diagramm erstellt wird. Auf diese Weise wird z.B. beim Erstellen einer Verbindung im Diagramm eine Referenz gesetzt. Falls für eine Referenz im Metamodell eine Verbindung im Diagramm beschrieben wurde, generiert EuGENia automatisch den notwendigen Code der diese Referenz verwaltet. An einigen Stellen im Modell reicht dies jedoch nicht aus. Zur Umsetzung der in Abschnitt 3.1.2 beschriebenen Ereignissteuerung ist es notwendig, dass beim Erstellen einer Verbindung zwischen TaskChannel und TaskInput in beiden Elementen eine Referenz auf das jeweils andere gesetzt wird. Der TaskChannel benötigt eine Liste aller Empfänger, der TaskInput benötigt einer Referenz auf den Channel, um im Code-Generator den richtigen Datentyp für den Empfänger-Task bereitzustellen. Zur Implementierung solcher Referenzen, bei denen mehrere Elemente verändert werden, müssen eigene Kommandos zur Erstellung, Bearbeitung und zum Löschen erstellt werden. Wichtig dabei ist, dass auch beim Verändern der Verbindung die Referenzen angepasst werden.

Neben solchen Änderungen am Modell bei der Bearbeitung eines Diagramms, muss das Modell auch bei der Initialisierung bearbeitet werden. Mit EuGENia wurde ein Wurzelelement-Type für jedes Diagramm definiert. Beim Erstellen eines Diagramms muss ein Element dieses

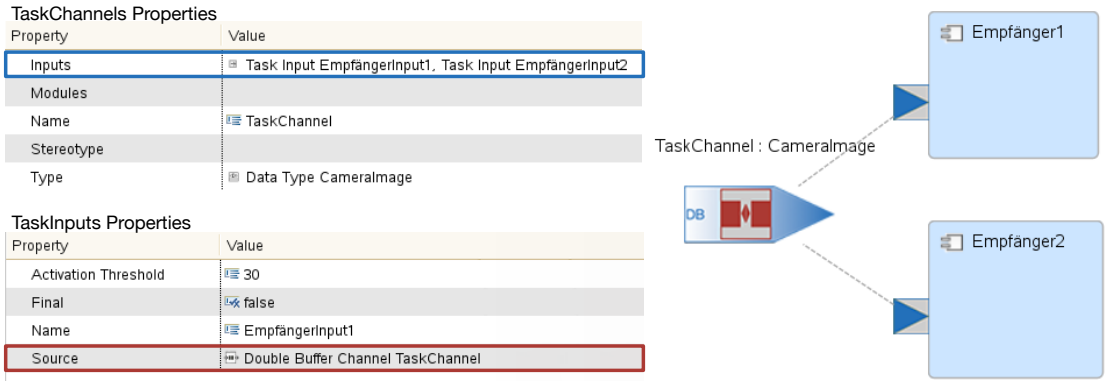


Abbildung 26: Screenshots des Diagrammeditors. Wird im Diagramm eine Relation von TaskChannels zu TaskInputs über das Erstellen einer Verbindung gesetzt, wird im referenzierten Objekt eine Rückwerts-Relation erstellt.

Typs angegeben werden, um zu beschreiben, welcher Teil des Modells bearbeitet wird. Falls es noch kein Modell gibt, wird ein Modell mit dem Diagramm-Wurzelelement als Modell-Wurzelelement erstellt. Dieses Verhalten kann jedoch zu Problemen führen: Wird als erstes Diagramm z.B. ein Parameter-Diagramm erstellt, würde das Wurzelelement des Modells ein TaskingModule sein. In ein solches Modell könnten keine weiteren Module eingefügt werden und der Code-Generator könnte mit der falschen Modellstruktur nicht umgehen. Um dieses Problem zu lösen wurden alle Diagramme so angepasst, dass sie ein valides Modell mit einer TaskingEnvironment als Modellwurzelelement erstellen. Bei einigen Diagrammen war es auch notwendig Wrapper-Elemente zu erstellen, in die die Diagramm-Inhalte eingefügt werden. Wie in Abschnitt 4.2.1 beschrieben, müssen auch solche selbst implementierten Veränderungen am Modell in Transaktionen eingefügt werden, um sicherzustellen das dieses konsistent bleibt.

4.3.5 Validierung mit EVL und OCL

Eine der Anforderungen an den Diagrammeditor war, eine generische Validierung. Das GMF lässt eine Erstellung von OCL-Constraints, wie in Abbildung 18 mit UML gezeigt, auch für eigene Elemente zu [Ebe14]. Auch wenn OCL eine sehr mächtige Syntax bietet, hat ein Validierung mit der Sprache einige Begrenzungen [Dim15]. Schlägt eine OCL-Regel fehl, wird dem Nutzer nur eine Fehlermeldung mit dem Namen des Constraints angezeigt. Zudem können OCL-Constraints keine anderen aufrufen. Auch Warnungen und Möglichkeiten zur

```
1  context Attribute{
2   constraint NotAbstract {
3    guard : self.satisfies('HasType')
4
5    check {
6            var ret : Boolean;
7            if(self.type.isTypeOf(DataType) or self.type.isTypeOf(Class)){
                 ret = not self.type.isAbstract;}
8            else{ret = true;}
9            return ret;}
10   message : 'Element ' + self.name + ' has abstract type ' + self.type.
         name + '!'
11  }
12 }
```

Quellcode 2: EVL-Quellcode-Beispiel einer Regel an Attribute-Elemente. Bevor die eigentliche Überprüfung der Regel ausgeführt wird, kann sichergestellt werden, dass andere erfüllt sind. Mit „check“ wird die eigentliche Rahmenbedingung beschrieben, „message“ gibt die Fehlermeldung an.

Fehlerbehebung werden nicht unterstützt.

Eine Alternative zu OCL ist die Epsilon Validation Language (EVL). Unabhängig von EuGENia können mit dieser Sprache Rahmenbedingungen an ein Modell und GMF Diagramme erstellt werden. EVL unterstützt Warnungen, Fehlerbehebungen und es können Bedingungen an die Überprüfung der Constraints gestellt werden. So ist es möglich eine Regel nur auszuführen, wenn eine andere vorher erfolgreich validiert wurde oder das Kontext-Element sich an einer bestimmten Stelle im Modell befindet. Wie in Quellcode 2 dargestellt, ist es auf diese Weise möglich, nur dann zu prüfen, ob der Typ eines Attributs abstrakt ist, falls dieser gesetzt ist. Die Sprache ermöglicht auch eine Überprüfung von Attributen, die nicht in allen Unterklassen enthalten sind. Beispielsweise kann der Typ eines Attributs auch ein Basisdatentyp, ohne den Parameter „isAbstrakt“ sein. Um dem Nutzer möglichst präzise Informationen über den Fehler im Modell bereitzustellen, können eigene Fehlermeldungen erstellt werden. Die Validierung wird über ein eigenes Eclipse-Plugin implementiert und verwendet standardmäßig die Eclipse-Fehleranzeige. Abbildung 27 zeigt, wie Fehler des Modells im Editor angezeigt werden. Neben dem Diagrammedtior werden auch andere Plugins über die Fehler benachrichtigt.

Um sicherzustellen, dass mit einem Modell Quellcode generiert werden kann, wird das Modell ausführlich überprüft. Neben der Validierung, ob modellierte Parameter des Tasking

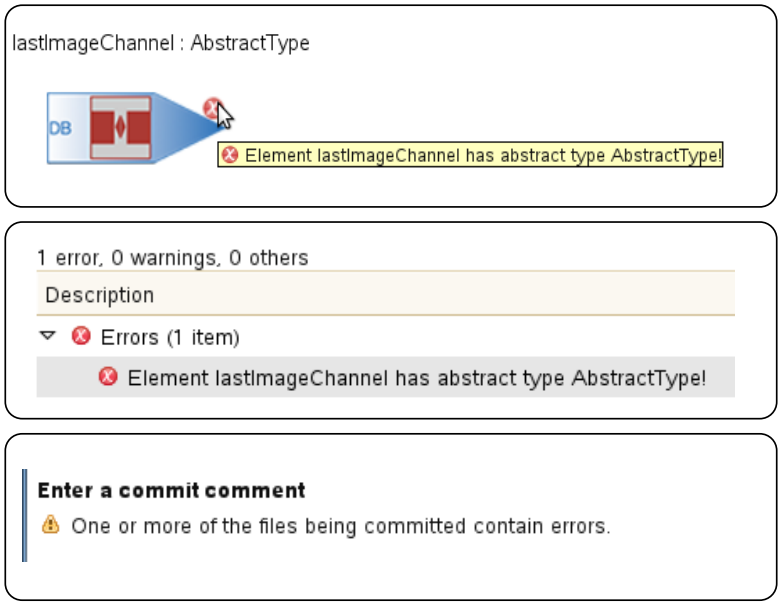


Abbildung 27: Darstellung der Validierung im Editor. Die mit EVL definierte Fehlermeldung wird an mehreren Stellen angezeigt. Da das GMF die Eclipse-Fehlerbehandlung verwendet, können auch externe Plugins bei Fehlern waren. Der untere Screenshot zeigt die Warnung des Subversion-Plugins beim committen fehlerhafter Diagramme oder Modelle.

Frameworks nur korrekte Werte enthalten, wird auch überprüft, ob die Zahl der lesenden und schreibenden Tasks eines Channels korrekt sind. Außerdem wird überprüft, ob für alle Elemente ein eindeutiger Name gesetzt ist und Warnungen ausgegeben falls diese nicht den Konventionen entsprechen. Falls ein Klassenattribut beispielsweise mit großem Buchstaben beginnt, wird der Nutzer gewarnt und eine automatische Korrektur vorgeschlagen.

Bei dem UML-Editor Papyrus muss eine Validierung des Modells durch den Nutzer ausgelöst werden. Da es für den Nutzer jedoch aufwendig ist, nach jeder Änderung die Validierung auszuführen, wird dieser Vorgang auf Dauer nicht verwendet werden. Um den Vorgang in dem Editor zur Sprache des Tasking Frameworks zu optimieren, wurde die Validierung so implementiert, dass sie automatisch beim Speichern eines Diagramms ausgeführt wird. Eine Validierung nach jeder Änderung im Diagramm wäre auch möglich, allerdings würde der Nutzer so durch viele Fehlermeldungen frustriert werden. Da z.B. ein TaskChannel, wenn er in das Diagramm eingefügt wird noch keinen Typ und Namen hat, würde die Validierung

direkt mehrere Fehler melden.

4.3.6 Ontologische Erweiterungen des Modells

Ein in Abschnitt 3.2 beschriebenes Konzept sieht Nutzerweiterung am Modell vor. Um eine Modellierung von speziellen Projektanforderungen zu ermöglichen, sollte es dem Nutzer möglich sein, eigene Modellelemente zu erstellen. Wie beschrieben muss eine Umsetzung dieses Konzepts aber verhindern, dass alle Elemente auf beliebige Weise erweitert werden können, da das Modell dann zu komplex werden kann und beliebige Erweiterungen vom Code-Generator nicht umgesetzt werden können. Zur Umsetzung der Nutzererweiterungen in der modellgetriebenen Entwicklung muss definiert werden, wie diese sich im Code auswirken. Für das Tasking Framework macht es Sinn, Nutzerweiterungen für TaskChannels und Module zuzulassen, da diese durch von durch TaskChannels oder Tasks abgeleitete Klassen um eigene Parameter erweitern werden können. Eine Implementierung eines TaskChannels könnte einen für das Projekt speziellen Mechanismus zur Speicherung der Daten erstellen. Da ein solcher Mechanismus Parameter hat, die sich je nach Instanz unterscheiden können, sollten diese Parameter auch ins Modell aufgenommen werden können.

Ein solcher Mechanismus zur Erweiterung der modellgetriebenen Entwicklung muss aus zwei Komponenten bestehen: Erstens eine Modellierungsumgebungen, die Erweiterungen zu lässt und zweitens einem Code-Generator, der mit den Erweiterungen umgehen kann. Eine reine Erweiterung des Modells um neue Elemente ohne Konzept, wie diese im weiteren Entwicklungsprozess verwendet werden, ist nicht sinnvoll. Aus diesem Grund wurde neben einer Sprache und einem Editor, die Erweiterungen unterstützen, auch ein Konzept erstellt, wie der Code-Generator die Erweiterungen verwendet.

Der Erweiterungsmechanismus der Sprache und des Editors orientieren sich an den ontologischen Erweiterungen von UML. Elemente die erweitert werden dürfen, können in einem Profil-Diagramm als Metaklasse eingefügt und durch Stereotypen erweitert werden. Im Gegensatz zu UML sind die Projekterweiterungen dabei nicht in ein getrenntes Projekt aufgeteilt. Dies hat den Hintergrund, dass projektübergreifende Erweiterungen des Tasking Framework in das Metamodell der Sprache eingebaut werden sollen. Erweiterungen an einem Projekt machen in dem Modell Sinn, da diese Änderungen dann direkt verwendet werden können, ohne das Erweiterungs-Modell für das Nutzermodell aktivieren zu müssen. Der implementierte

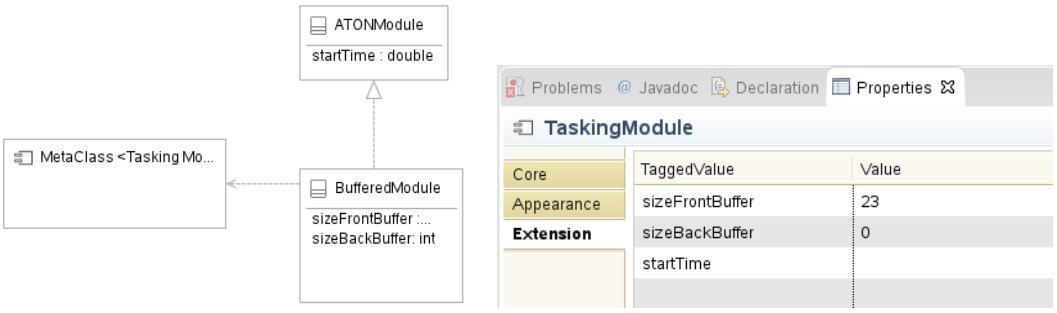


Abbildung 28: Screenshots des Erweiterungsmechanismus des Editors zur Modellierung. Links die Definition der Erweiterung, rechts die Ansicht zur Modellierung der erweiterten Parameter einer Instanz eines „BufferedModules“.

Editor bietet eine automatische Aktualisierung der Palette, zum Einfügen der Elemente in das Diagramm, sobald eine Erweiterung erstellt wurde.

Im Gegensatz zu UML ist ein weiterer Unterschied, dass ein Element immer nur durch einen Stereotypen erweitert werden kann. Diese eindeutige Zuordnung von erweitertem Modellelement zu Stereotyp ermöglicht dem Code-Generator eine Basisklasse für die zu implementierende Erweiterung bereitzustellen. Der Code-Generator erstellt eine leere Klasse, die von der Quellcode-Entsprechung des Basiselements erbt und den Namen des Stereotyps hat. Die generierte Klasse hat die erweiterten Elemente als Parameter im Konstruktor und kann auf diese Weise die Werte des Modells im generierten Code verwenden. Da die generierte Klasse von der Basisklasse erbt, ist die Verwendung des Elementes auch ohne zusätzliche Implementierung gültig, verwendet die Parameter allerdings nicht. Da Stereotypen als Classifier mit Mehrfachvererbung umgesetzt sind kann ein Modellelement dennoch die Erweiterungen mehrerer Stereotypen besitzen.

Neben dem Diagramm zur Beschreibung der Erweiterungen muss der Editor so angepasst werden, dass die in dem Diagramm beschriebenen Elemente erstellen und in den anderen Diagrammen einfügen kann. Zur Umsetzung mussten die Paletten der anderen Diagramme so implementiert werden, dass diese automatisch aktualisiert werden und angepasste Werkzeuge zur Erstellung des jeweiligen Elements verwenden. Jedes Element in der Palette eines Diagramms repräsentiert ein Werkzeug (engl. CreationTool) zur Erstellung des Diagrammelements. Normalerweise werden die Elemente dabei initialisiert und mit Standardwerten versehen. Zur Erstellung der erweiterten Modellelemente musste der Editor so angepasst werden, dass

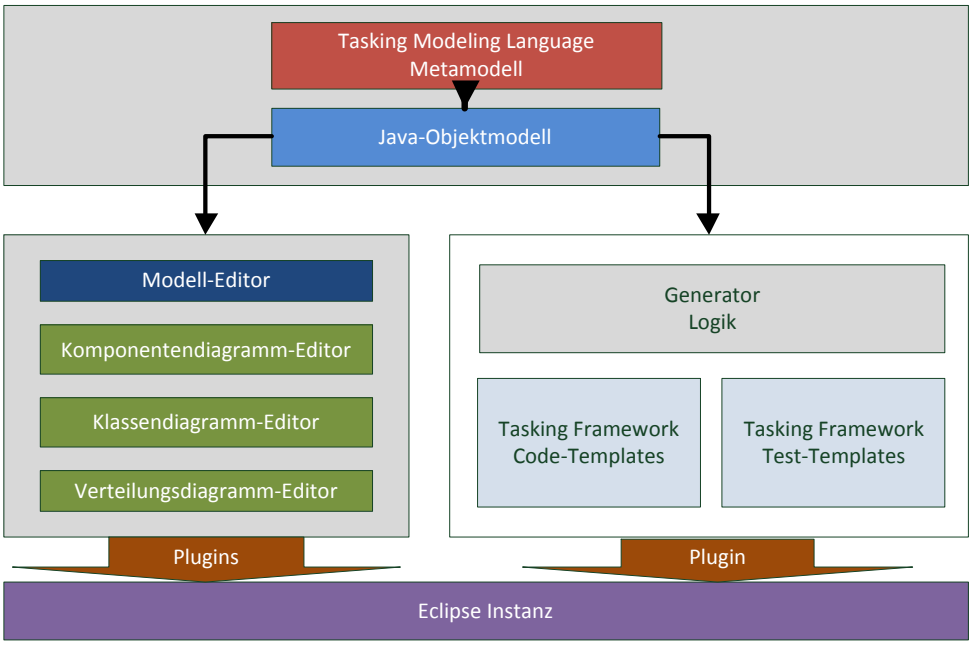


Abbildung 29: Darstellung des Aufbaus der Entwicklungsumgebung zur modellgetriebenen Entwicklung. Sowohl die Diagrammeditoren als auch der Code-Generator verwenden das Java-Objektmodell des Metamodells, um das Nutzermodell zu lesen. Neben den Diagrammeditoren wird auch der Code-Generator als Eclipse-Plugin bereitgestellt.

es möglich war eigene CreationTools zu erstellen. Diese erstellen jeweils das Basiselement und fügen automatisch den jeweilige Stereotyp hinzu. Zur optischen Anpassungen ist es möglich sowohl das Icon als auch das Diagrammelement selbst anzupassen. Neben optischen Anpassungen zeigt der Editor, wie in Abbildung 28 dargestellt, für angepasste Elemente eine zusätzliche Seite an in der die erweiterten Attribute gesetzt werden können.

4.4 Integration des Code-Generators

Mit einer Modellierungssprache und den Diagrammeditoren zur Erstellung des Modells ist erst der erste Schritt zur Umsetzung einer modellgetriebenen Entwicklung erfolgt. Wie in Abschnitt 2.1 beschrieben ist der nächste Schritt die Generierung von Quellcode aus dem Modell. Der existierende Code-Generator ist wie auch die Diagrammeditoren auf Basis des

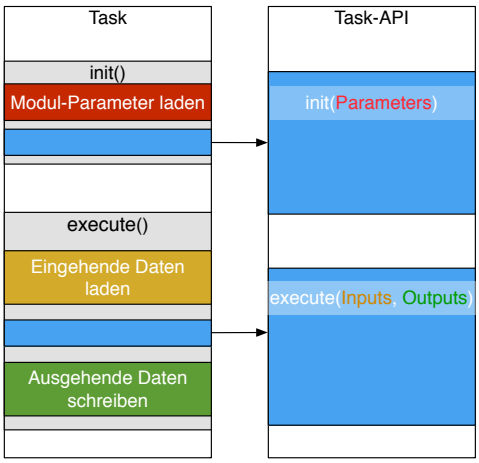


Abbildung 30: Durch die modellgetriebene Entwicklung wird es möglich den Code zum Laden der Parameter eines Moduls und der Verwaltung der ein- und ausgehenden Daten zu generieren. Um diesen Code vor dem Nutzer zu verbergen macht die Anpassung der API Sinn.

EMFs umgesetzt. Aus diesem Grund ist ein Zugriff des Code-Generators auf das Nutzermodell über die für die Diagrammeditoren erstellte Java-Implementierung des Metamodells möglich.

Da die Diagrammeditoren als Eclipse-Plugin bereitgestellt werden und somit zur Modellierung eine Instanz der Entwicklungsumgebung geöffnet ist, macht es Sinn den Code-Generator auch als Plugin in die Entwicklungsumgebung zu integrieren. Auf diese Weise ist es möglich über das Kontext-Menü des Nutzermodells eine Code-Generierung anzubieten. Im Zug der Weiterentwicklung des Entwicklungsumgebung macht auch die Erstellung eines Projekttypen für das Tasking Framework Sinn. In diesem kann automatisch ein leeres Modell mit den Basis-Diagrammen erstellt werden. Der aktuell sehr aufwendige Prozess der initialen Erstellung einer Software, die auf dem Tasking Framework basiert, ist so in wenigen Klicks über eine vorkonfigurierte Eclipse-Umgebung möglich.

Das wichtigste Element der aktuellen API der Tasks des Tasking Framework ist die „execute()“-Methode, die aufgerufen wird, sobald der Task ausgeführt werden soll. Da es aktuell noch keinen Standard-Mechanismus zur Verwaltung der Daten in den TaskChannels gibt, müssen die Daten manuell aus den Channels geladen werden. Da der Zugriff auf die TaskChannels jedoch aus dem Modell generiert werden kann, macht es Sinn die API mit dem Decorator-Pattern anzupassen. Die „execute()“-Methode kann soweit implementiert werden, dass die

Datentypen der ein- und ausgehenden Daten vorbereitet und an eine neue Funktion übergeben werden, in der auf die Daten über Funktionsparameter zugegriffen werden kann. Auf diese Weise kann auch die Initialisierung der Tasks aus einem generierten Konfigurationsmanager von dem Nutzer verborgen werden.

5 Bewertung und Exemplarische Anwendung der modellgetriebenen Entwicklungskonzepte

Um die entwickelte, grafische Modellierungssprache bewerten zu können wurde sie mit den vorher definierten Zielen verglichen und an einem Projekt des Tasking Frameworks exemplarisch angewendet. Aus den so gewonnenen Erfahrungen zur Modellierung mit der Sprache konnte dann die Bedeutung der Sprache für das Tasking Framework bewertet werden.

5.1 Vergleich mit den Anforderungen

In Abschnitt 1.2.3 wurde Ziele an die Sprache und Werkzeuge zur modellgetriebenen Entwicklung beschrieben. In diesem Kapitel wird überprüft, ob die Umsetzung diesen Zielen genügt. Mit der Sprache sollen alle Parameter des Tasking Frameworks modelliert werden können. In Abschnitt 3.1 wurden Anforderungen definiert, was modelliert werden muss, um eine Architektur einer auf dem Tasking Framework basierende Software zu beschreiben.

Die grafische Modellierungssprache zur Beschreibung des Tasking Frameworks wurde mit der Metasprache Ecore des EMFs erstellt und erfüllt somit die in Abschnitt 1.2.3.4 geforderte Kompatibilität zu dem bestehenden Code-Generator.

Alle Aspekte der Kommunikation und Ereignissteuerung des Tasking Framework sollten grafisch modelliert werden können. Mit dem in Abbildung 31 blau umrahmten Komponentendiagramm kann beschrieben werden, welche Tasks miteinander kommunizieren und die Ereignissteuerung des Tasking Framework über TaskChannels, TaskEvents und TaskInputs konfiguriert werden. Zur Beschreibung der Parameter eines Moduls und der Datentypen der TaskChannels, kann das Klassendiagramm (vergl. Abbildung 31, rechts) verwendet werden. Über das in Abbildung 31 braun umrahmte Verteilungsdiagramm können Tasks zu TaskSets zugeordnet werden. Mit diesen drei Diagrammen können alle der in Abschnitt 3.1 dargestellten Aspekte, beschrieben werden. Zusätzlich haben alle Diagrammelemente noch Eigenschaften, die im Editor gesetzt

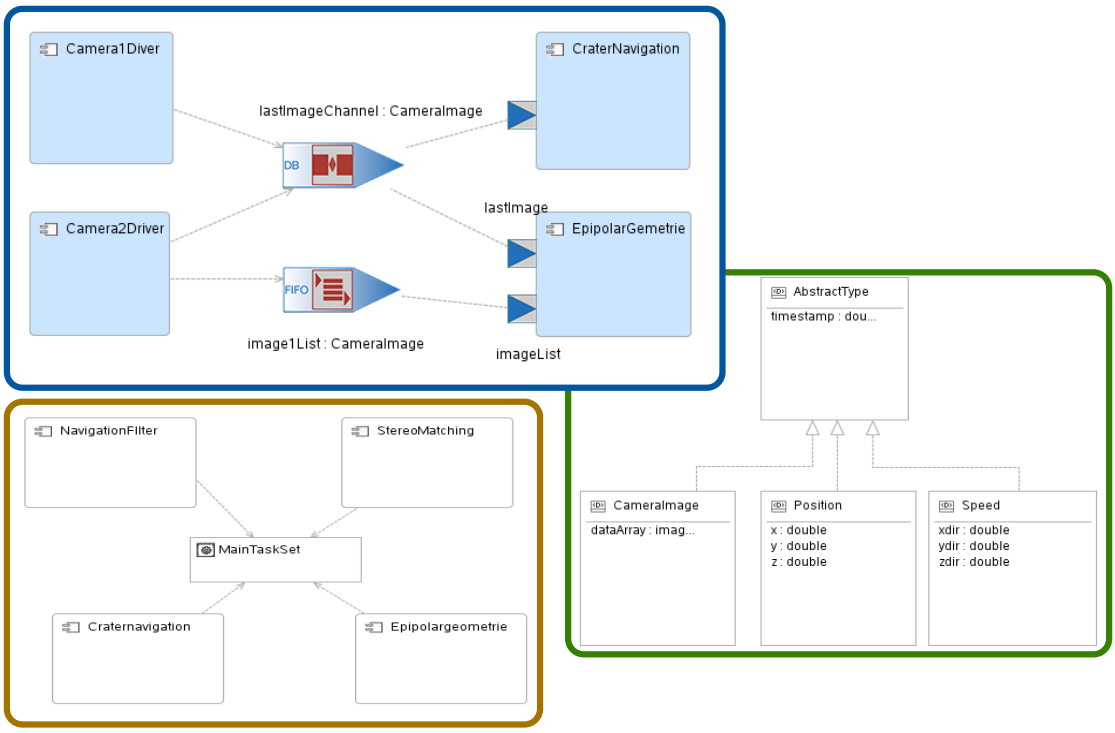


Abbildung 31: Screenshots des Editors bei der Darstellung eines Komponentendiagrams zur Modellierung der Kommunikation (oben), eines Klassendiagramms zur Beschreibung der Datentypen und Parameter (rechts) und eines Verteilungsdiagramms zur Zuordnung der Tasks zu den TaskSets (unten).

werden können.

Neben den Anforderungen an die Sprache selbst, beschreibt Abschnitt 1.2.3.3 auch, dass ein Editor zur grafischen Modellierung zur Verfügung stehen soll. Dieser darf nur wohlgeformte Diagrammkombinationen zulassen und eine Validierung der Elementeigenschaften ermöglichen. Abbildung 32 zeigt die Unterstützungs-Mechanismen des Editors. Dieser bietet sowohl eine Validierung als auch eine kontextsensitive Darstellung ob ein Element erstellt werden kann oder nicht. Zusätzlich werden auch Vorschläge zur Erstellung von Elementen geliefert. Außerdem ermöglicht der Diagrammeditor eine Bearbeitung des Modells in mehreren Diagrammen gleichzeitig und er bietet die Möglichkeit neue Elemente zu definieren.

Da für die Sprache ein Editor erstellt wurde, der nur gültige Diagramme zulässt und den Nutzer auf Fehler hinweist, den Anforderungen also entspricht, die Modellelemente alle Parameter

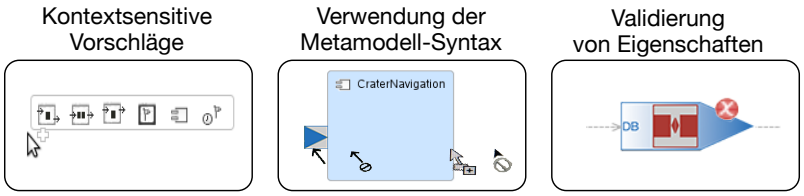


Abbildung 32: Darstellung der Unterstützungs-Mechanismen durch den Editor. Der Editor macht je nach Position der Maus Vorschläge welche Elemente eingefügt werden können. Wird ein Element aus der Palette ausgewählt, zeigt der Editor an, wo dieses eingefügt werden kann und wo nicht. Die Eigenschaften von eingefügten Elementen werden validiert.

des Tasking Framework enthalten und die Diagramme den Aufbau und Kommunikation der Software vollständig beschreiben, wurden alle Ziele an die grafische Modellierungssprache erfüllt.

5.2 Exemplarische Anwendung

Um die Sprache an einem Beispiel zu testen, wurde die Software des Projekts ATON modelliert. Da der existierende Code-Generator für dieses Projekt entwickelt wurde, kann die entwickelte Sprache so mit dem vorher verwendeten UML-Modell verglichen und untersucht werden, welchen Einfluss dies auf den Code-Generator hat.

Im Unterschied zu den vorher verwendeten Komponenten zur Code-Generierung in ATON wurde die Sprache nicht für dieses Projekt sondern allgemein für Projekte des Tasking Frameworks entwickelt. Da ATON eine spezielle Form eines TaskChannels verwendet, musste hier der Erweiterungsmechanismus verwendet werden. Mit dem Modell der neu definierten Tasking Modeling Language konnten alle in UML enthaltenen Parameter des Tasking Framework modelliert und somit der gesamte Code zur Kommunikation der Softwaremodule untereinander generiert werden. Außerdem kann auch ein Konfigurationsmanager zur Initialisierung der Tasks aus den modellierten Parametern generiert werden.

Da vorher kein Editor zur Modellierung des Tasking Frameworks zur Verfügung stand, wurden fertige UML-Editoren verwendet. Bei der Modellierung gab es keinerlei Einschränkungen, der Nutzer konnte beliebige Elemente in das Diagramm ziehen. Auch die Eigenschaften der

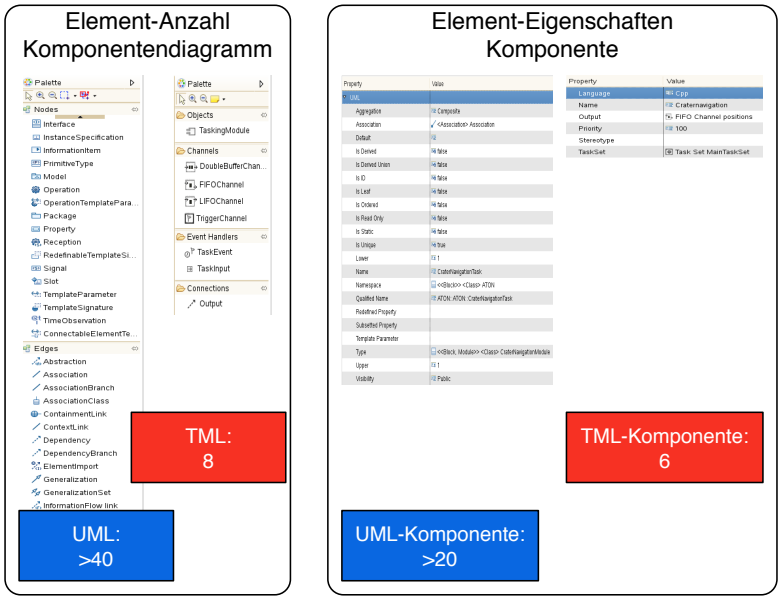


Abbildung 33: Vergleich der Komplexität der Modellierungssprachen. Links ein Vergleich der Anzahl an Elementen, die in ein Komponentendiagramm eingefügt werden können, rechts die Anzahl der Eigenschaften einer Komponente.

Elemente wurden nicht validiert. Die Modellierung konnte also nur von Entwicklern, die die internen Abläufe des Code-Generators kennen, vorgenommen werden. Mit den in Abbildung 32 gezeigten Unterstützungen des neuen Editors wird die Modellierung effektiv vereinfacht und kann nun ohne Kenntnisse von Code-Generator oder Tasking Framework verwendet werden, da der Nutzer direkt auf alle Falscheingaben hingewiesen wird.

Im Editor wird auch ein Vorteil der eigene Sprache gegenüber einer Erweiterung aus UML und anderen Sprachen sichtbar: In Abbildung 33 ist gezeigt, dass in das Komponentendiagramm eines UML-Editors über die fünffache Menge an Elementen im Vergleich zu dem selbst definierten Komponentendiagramm eingefügt werden können. Auch die Elemente selbst enthalten das Vielfache an Eigenschaften im Vergleich zu den Elementen der Sprache zur Beschreibung das Tasking Framework. Da in diese nur die wirklich benötigten Elemente und Eigenschaften eingebaut werden müssen ist die Sprache deutlich einfacher. Es gibt weniger Fehlerquellen und der Nutzer muss weniger nach dem gewünschten Element suchen.

Mit dem in Abschnitt 1.2.3.2 beschriebenen Code-Generator wurde Quellcode aus einem UML-Modell generiert. Um mit einem UML-Modell Code generieren zu können, war sehr viel Logik

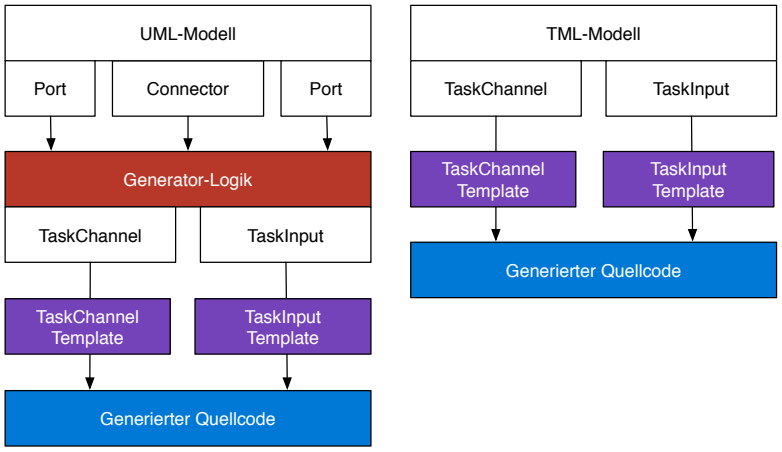


Abbildung 34: Vergleich der Code-Generatoren mit UML-Modell und einem Modell der neu definierten Tasking Modeling Language. Mit dem UML-Modell-Generator war viel Logik notwendig, um aus den UML-Elementen die Informationen für die Templates des Tasking Frameworks zu sammeln. Diese Logik wird mit einem Modell, das speziell für das Tasking Framework entwickelt wurde nicht mehr benötigt.

im Generator notwendig, um die in den generischen UML-Elementen enthaltenen Informationen für die Templates des Tasking Framework bereitzustellen. Im Code-Generator wurden Klassen für die Elemente des Tasking Frameworks erstellt um dort die für die Templates notwendigen Informationen zu speichern. Die Elemente der neu definierten Sprache entsprechen bereits den Elementen des Tasking Frameworks und enthalten alle für die Templates notwendigen Informationen. Wie in Abbildung 34 anhand der Verbindungen zwischen den Tasks dargestellt, kann die Logik zur Übersetzung von UML in eine für das Tasking Framework kompatible Sprache auf diese Weise weg fallen. Abbildung 35 zeigt, dass alleine auf diese Weise der notwendige Code des Generators um ca. 30% verkürzt werden kann. Da besonders die Logik des Code-Generators schwer anzupassen ist, bedeutet die Reduktion dieses Anteils um fast 80% einen deutlichen Vorteil für die Wartbarkeit des Generators. Neben der Sprache und dem Editor wird durch die Entwicklung einer speziell für das Tasking Framework entwickelten Sprache also auch der Code-Generator sehr viel einfacher. Anpassungen und Erweiterungen daran werden so leichter möglich.

Die Parameter des Tasking Framework im alten UML-Modell konnte nicht validiert werden. Um häufige Fehler zu vermeiden, wurden in den Code-Generator einige Überprüfungen eingebaut,

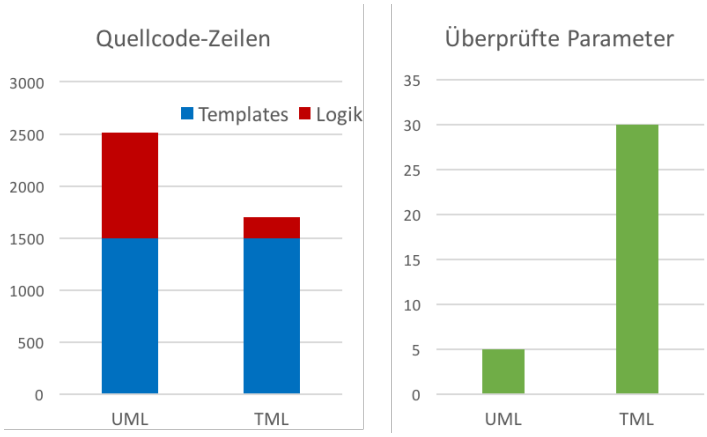


Abbildung 35: Quellcode-Zeilen des Generators und überprüfter Parameter bei der Verwendung des alten UML-Modells im Vergleich zu einem Modell der entwickelten Sprache. Der Code-Generator benötigt weniger Code, es werden jedoch deutlich mehr Parameter überprüft.

deren Implementierung jedoch im Vergleich zu den EVL-Constraints sehr aufwendig ist. Um alle möglichen Fehler eines Parameters im Source-Code abzufangen, müssen mehrere Überprüfungen vorgenommen werden, während eine Beschreibung eines Constrainst nur wenige Zeilen Code sind. Die Validierung im Modell hat also neben der direkten Rückmeldung an den Nutzer auch noch den Vorteil, dass der Code-Generator weiter vereinfacht werden kann. Wie in Abbildung 35 gezeigt, ist es mit einer Kombination aus TML-Modell und neuem Code-Generator auf diese Weise möglich, deutlich mehr Parameter zu überprüfen, den Code-Generator jedoch sehr klein zu halten.

5.3 Bedeutung für das Tasking Framework

Die modellgetriebene Entwicklung wurde aufbauend auf dem existierenden Code-Generator des Projekts ATON entwickelt. Da in dem Projekt jedoch nicht alle Funktionen des Tasking Frameworks genutzt werden, erweitert die neu entwickelte Sprache sowohl die Modellierung als auch die Code-Generierung um zahlreiche Möglichkeiten. In ATON werden keine TaskEvents benötigt und es gibt zu jedem TaskChannel nur genau einen schreibenden Task. Mit der grafischen Modellierungssprache können nun Projekte erstellt werden die alle Funktionen des Tasking Frameworks verwenden.

Durch die Umsetzung der modellgetrieben Entwicklung wird es möglich ein neues, auf dem Tasking Framework beruhendes Projekt, innerhalb von wenigen Minuten zu erstellen. Der vorher sehr aufwendige Prozess der initialen Erstellung von mehreren Tasks und der Implementierung der Kommunikation kann nun in wenigen Diagrammen beschrieben und der notwendige Code generiert werden.

Auch während dem Projektverlauf ist es ohne großen Aufwand möglich neue Komponenten in die Software einzufügen oder verwendete Kommunikationswege und Daten anzupassen. Die Entwicklung an entsprechenden Projekten kann so deutlich dynamischer werden.

Ein weiterer Vorteil ist die Tatsache, dass die Verwendung des Tasking Frameworks durch Modellierung sehr viel einfacher wird. Auch Entwickler, die die internen Abläufe des Frameworks nicht kennen, können dieses so verwenden. Durch die starke Unterstützung und Validierung des Editors kann der Nutzer so nichts falsch machen.

6 Fazit und Ausblick

Im Rahmen dieser Arbeit wurde erfolgreich eine grafische Modellierungssprache für die Code-Generierung eines ereignisgesteuerten Echtzeit-Laufzeitsystems entwickelt. Um die optimale Sprache zur Beschreibung einer solchen Software zu finden, wurde sich mit den Konzepten einer modellgetriebenen Softwareentwicklung auseinandergesetzt und anhand dieser entschieden, dass die Verwendung einer speziell für das Aufgabengebiet entwickelten Sprache sinnvoller ist, als die Anpassungen von vorhandenen Sprachen wie UML, SysML oder AADL. Mit den Diagrammen vorhandener Sprachen ist eine vollständige Beschreibung des Systems möglich, allerdings werden durch deren generische Syntax auch für das Tasking Framework ungültige Anordnungen zugelassen. Einschränkungen der Sprachen über z.B. OCL-Constraints bei UML ermöglichen es solche Anordnungen auszuschließen, allerdings nur in einer nachträglichen Validierung. Eine Beschreibung aller notwendigen Constraints ist zudem sehr aufwendig und fehleranfällig.

Da die Sprache speziell für den Anwendungsfall der ereignisgesteuerten Kommunikation entwickelt wurde, ist sie zudem sehr viel einfacher, leichter zu verstehen und der modellierende Nutzer hat weniger Möglichkeiten Fehler zu machen.

Die entwickelte Sprache enthält mehrere Diagramme, die die zu modellierende Software unter mehreren Geschichtspunkten beschreiben. Zur Erstellung eines Editors wurden verschiedene Werkzeuge verglichen und schließlich eine Kombination aus dem Graphical Modeling Framework und dem darauf aufbauenden Eugeina verwendet. Der Editor lässt nur für das Tasking Framework gültige Konstrukte zu und ermöglicht eine weitere Validierung der Diagramme.

Ein wichtiges Konzept der modellgetriebenen Entwicklung ist die Erweiterbarkeit und Möglichkeiten zur Anpassung der Sprache an geänderte Anforderungen. Neben der Anpassung des Metamodells zur Erweiterung von projektübergreifenden Elementen, bietet die Sprache auch eine Möglichkeit, projektspezifische Änderungen über ontologische Erweiterungen umzusetzen. Auf diese Weise erstellte, neue Elemente können im Editor direkt in den Diagrammen verwendet werden. Da die Modellierung nur der erste Schritt der modellgetriebenen Entwicklung ist, wurde der Mechanismus zur Erweiterung der Sprache so entwickelt, dass auch der Code-Generator mit den Erweiterungen umgehen kann.

Ausblick

Ein wesentlicher Bestandteil der modellgetriebenen Softwareentwicklung ist die Entwicklungsumgebung. Mit dieser werden sowohl die Diagramme erstellt als auch Code generiert. Auf Grund der großen Flexibilität und Erweiterbarkeit von Eclipse lässt sich so die Modellierung noch weiter vereinfachen. Neben Projekttypen und Perspektiven für das Tasking Framework kann auch der Diagrammeditor weiter verbessert werden. Ein Doppelklick auf eine Modul kann z.B. das Diagramm zur Beschreibung der Parameter dieses Moduls öffnen.

Da das Tasking Framework so weiterentwickelt werden soll, dass auch Software-Module verschiedener Sprachen kombiniert werden können, wurde die grafische Modellierungssprache darauf vorbereitet. Der nächste Schritt ist die Entwicklung eines Code-Generators der auch Module in Sprachen wie Ada direkt einbinden kann. Insbesondere hier ist eine Auseinandersetzung mit ESA Projekten wie CORDET [P&] und TASTE [PCD12] sinnvoll, da diese bereits die Möglichkeit bieten, verschiedene Sprachen zu kombinieren. Dort definierte Schnittstellen sollten auch für das Tasking Framework genutzt werden, um im besten Fall Softwaremodule direkt austauschen zu können.

Mit der modellgetriebenen Entwicklung für das Tasking Framework wurde ein neuer Entwicklungsprozess für zukünftige Projekte geschaffen. Aus diesem Grund macht die Auseinandersetzung mit Funktionen zur Verbesserung einer solchen Software-Entwicklung Sinn. Neben der Generierung von produktivem Quellcode für die Projekte könnten auch Unit-Tests generiert werden, mit denen die generierte Software während des Entwicklungsprozess automatisiert getestet werden kann. Mit solchen Tests auf Systemebene könnte der generierte Code direkt nach einer initialen Erstellung eines Projekts ausgeführt und im Verlauf der Entwicklung überprüft werden, wie Änderungen an den Software-Modulen sich auf das Projekt auswirken. Neben Unit-Tests könnten auch Build-Dateien und Dokumentationen generiert werden.

Zusätzlich zu diesen Möglichkeiten zur Weiterentwicklung der modellgetriebenen Entwicklung des Tasking Frameworks macht auch die Integration des Code-Generators zur Erstellung mehrerer Diagramme mit EuGENia in die Epsilon-Software Sinn. Die in dieser Arbeit implementierte Funktion mehrere Diagramme aus einem Metamodell generieren zu können bietet eine deutliche Aufwertung für EuGENia, da die Erstellung vieler grafischer DSLs so sehr beschleunigt werden kann.

Anhang A: Vollständiges Ecore-Metamodell

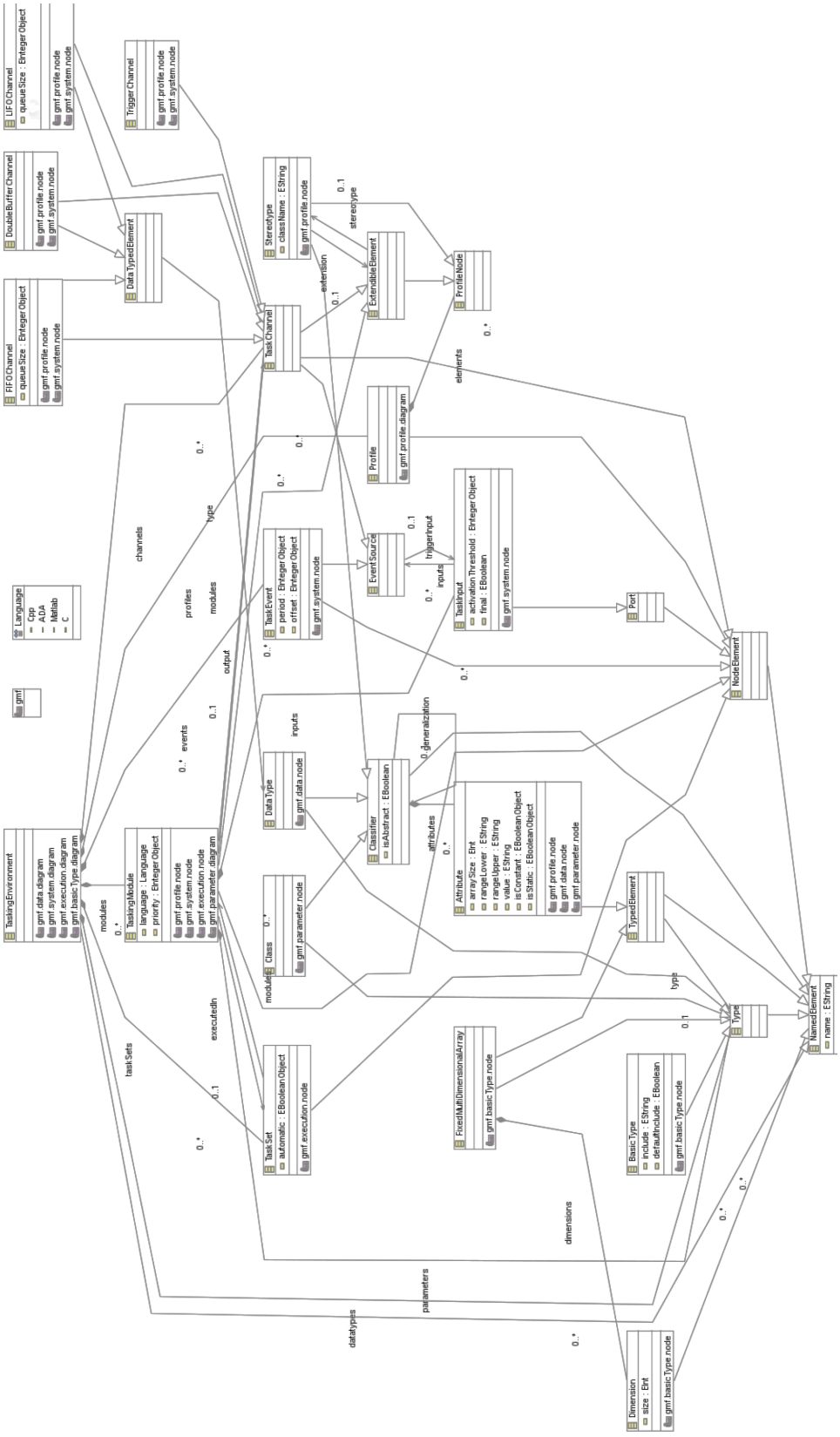


Abbildung 36: Vollständiges Ecore-Diagramm des Metamodells der Tasking Modeling Language.

Literaturverzeichnis

[AADa] AADL Wiki. Osate 2 - AadlWiki. URL: https://wiki.sei.cmu.edu/aadl/index.php/Osate_2.

[AADb] AADL Wiki. Updating the Meta-Model - AadlWiki. URL: https://wiki.sei.cmu.edu/aadl/index.php/Updating_the_Meta-Model.

[ABDJ05] Anas Abouzahra, Jean Bézivin, Marcos Didonet Del Fabro, and Frédéric Jouault. *A practical approach to bridging domain specific languages with UML profiles*. 2005.

[AK03] Colin Atkinson and Thomas Kühne. *Model-Driven Development: A Metamodeling Foundation*. IEEE Software, 2003.

[AK07] Colin Atkinson and Thomas Kühne. *Reducing accidental complexity in domain models*. Springer, 2007. doi:10.1007/s10270-007-0061-0.

[AW04] Kendall Scott Axel Uhl Stephen J. Mellor and Dirk Weise. *Model-Driven Architecture*. Springer Verlag, 2004.

[BGK+10] By Christian Brand, Matthias Gorning, Tim Kaiser, Jürgen Pasch, and Michael Wenz. Graphiti. pages 1–10, 2010.

[Bro04] W. Brown. *Model Driven Architecture: Principles and Practice*. Springer-Verlag, 2004.

[Car] Carnegie Mellon University. AADL Ressources. URL: http://www.aadl.info/aadl/currentsite/tool/metamod.html.

[CGS12] Hyun Cho, Jeff Gray, and Eugene Syriani. *Creating visual Domain-Specific Modeling Languages from end-user demonstration* . IEEE, 2012. doi:10.1109/MISE.2012.6226010.

[Com12] Benoit Combemale. *A Tutorial about Metamodeling Using OMG Norms and Eclipse Modeling*. 2012. URL: http://www.combemale.fr/mde.

[con08] MISRA consortium. *Guidelines for the use of the C language in critical systems*.

MIRA Limited, 2 edition, 2008.

[CS] M T Chitra and Elizabeth Sherly. *Refactoring Sequence Diagrams for Code Generation in UML Models*.

[DALR15] Dimitrios S. Kolovos, Antonio García-Domínguez, Louis M. Rose, and Richard F. Paige. *Eugenia: towards disciplined and automated development of GMF-based graphical model editors*. Springer Berlin Heidelberg, 2015.

[Del] Julien Delange. Code Generation with AADL: A State-of-the-Art Report ≫ SEI Blog. URL: https://blog.sei.cmu.edu/post.cfm/code-generation-aadl-279.

[Dem] Sébastien Demathieu. *Tutorial OMG UML Marte*.

[DGD06] Dragan Djurić, Dragan Gašević, and Vladan Devedžić. *The Tao of modeling spaces*, volume 5. 2006. doi:10.5381/jot.2006.5.8.a4.

[DH12] Pierre De Saqui-Sannes and Jérôme Hugue. *Combining SysML and AADL for the Design, Validation and Implementation of Critical Systems*. 2012. URL: https://hal-univ-tlse3.archives-ouvertes.fr/hal-00669391/document.

[Dim15] Richard Paige Dimitris Kolovos, Louis Rose, Antonio García-Domínguez. *The epsilon Book*. Eclipse Foundation, 2015.

[DLRa] DLR. Webpräsens der Abteilung Software für Raumfahrtsysteme und interaktive Visualisierung. URL: http://www.dlr.de/sc.

[DLRb] DLR. Webpräsens des DLRs. URL: http://www.dlr.de/.

[DLRc] DLR. Webpräsenz der Einrichtung Simulations- und Softwaretechnik. URL: http://www.dlr.de/sc/.

[DLRd] DLR. Webpräsenz des Projekts ATON. URL: http://www.dlr.de/sc/desktopdefault.aspx/tabid-6965/11517_read-26855/.

[Dr.06a] Martin Dr. Glinz. *Einführung in die Modellierung*. 2006. URL: https://files.ifi.uzh.ch/rerg/arvo/ftp/inf_II/inf_II_kapitel_01.pdf.

[Dr.06b] Martin Dr. Glinz. *Metamodelle*. 2006. URL: https://files.ifi.uzh.ch/rerg/arvo/ftp/inf_II/inf_II_kapitel_13.pdf.

[DvIL12] Vladimir Dimitrieski, Milan Čeliković, Vladimir Ivančević, and Ivan Luković. *A Comparison of Ecore and GOPPRR through an Information System Meta Modeling Approach*. 2012.

[Ebe14] Moritz Eberhard. OCL in Eclipse EMF and GMF for model validation, 2014. URL: http://meberhard.me/use-ocl-eclipse-emf-gmf-model-validation/.

[Ecla] Eclipse Foundation. EMF Documentation. URL: http://www.eclipse.org/modeling/emf/docs/.

[Eclb] Eclipse Foundation. Emfatic. URL: http://www.eclipse.org/emfatic/.

[Eclc] Eclipse Foundation. GMFGen - Eclipsepedia. URL: https://wiki.eclipse.org/Graphical_Modeling_Framework/Models/GMFGen.

[Ecld] Eclipse Foundation. Graphiti Home. URL: https://eclipse.org/graphiti/.

[Ecle] Eclipse Foundation. SharedEditingDomain - gmftools. URL: http://code.google.com/p/gmftools/wiki/SharedEditingDomain.

[Eclf] Eclipse Foundation. Webpräsenz des Eclipse Modeling Frameworkse. URL: https://www.eclipse.org/modeling/emf/.

[Eclg] Eclipse Foundation. Webpräsenz Graphical Modeling Framework. URL: http://www.eclipse.org/modeling/gmp/.

[Eclh] Eclipse Foundation. Xtext - Language Development Made Easy! URL: https://eclipse.org/Xtext/.

[Epsa] Epsilon. Customizing a GMF editor generated by EuGENia. URL: http://www.eclipse.org/epsilon/doc/articles/eugenia-polishing/.

[Epsb] Epsilon. Inspecting EMF models with Exeed — Epsilon Weblog on WordPress.com. URL: https://epsilonblog.wordpress.com/2008/07/17/inspecting-emf-models-with-exeed/.

[ESA] ESA. SOCIS - AADL Designer: The world of the Eclipse Modeling Framework. URL: http://onad2012.blogspot.de/2012/08/the-world-of-eclipse-modeling-framework.html.

[Faka] Kirill Fakhroutdinov. UML Communication Diagrams. URL: http://www.

uml-diagrams.org/communication-diagrams.html.

[Fakb] Kirill Fakhroutdinov. UML Component Diagram. URL: http://www.uml-diagrams.org/component-diagrams.html.

[Fei] Peter Feiler. OSATE Plug-in Development. Software Engineering Institute. URL: http://www.aadl.info/aadl/currentsite/downloads/osateworkshop-72005.pdf.

[Fei04] Peter Feiler. *Plug-in Development for OSATE OSATE Plug-in Development Workshop*. 2004.

[FFVM04] Lidia Fuentes-Fernández and Antonio Vallecillo-Moreno. *An Introduction to UML Profiles*, volume 6. 2004.

[FKBG] Fabio Filippelli, Steffen Kollosche, Michael Bauer, and Markus Gerhart. Concepts for the model-driven generation of graphical editors in Eclipse by using the Graphiti framework. pages 1–14.

[Fou] Eclipse Foundation. Eclipse Modeling Framework. URL: https://www.eclipse.org/modeling/emf/.

[Fra14] Tobias Franz. *Modellbasierte Techniken zur Umsetzung von Software-Kommunikations-Mechanismen*. Number September. 2014.

[GÍ1] Sébastien Gérard. *Papyrus User Guide* . Eclipse Foundation, 2011.

[GDTS10] Sebastien Gerard, Cedric Dumoulin, Patrick Tessier, and Bran Selic. *Papyrus: A UML2 Tool for Domain-Specific Language Modeling*. Springer Berlin Heidelberg, 2010.

[GH10] Olivier Gilles and Jérôme Hugues. *Expressing and enforcing user-defined constraints of AADL models*. 2010. doi:10.1109/ICECCS.2010.53.

[GHvL+07] Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, David Hutchison, Josef Kittler, Jon M Kleinberg, Alfred Kobsa, Friedemann Mattern ETH Zurich, Switzerland C John Mitchell, Moni Naor, Oscar Nierstrasz, C Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, and Gerhard Weikum. *Model-Based Engineering of Embedded Real-Time Systems*. 2007. URL: http://rd.springer.com/chapter/10.1007/978-3-642-16277-0_19.

[Gra] Andreas Graf. Generation gap pattern, ecore, Graphiti and nice pieces of mwe/Xtext — 5ise. URL: http://5ise.quanxinquanyi.de/2012/11/09/generation-gap-pattern-ecore-graphiti-and-nice-pieces-of-mwextext/.

[IKS13] Muzaffar Iqbal, Muhammad Uzair Khan, and Muhammad Sher. *System Analysis and Modeling Using SysML*, volume 215. 2013. URL: http://link.springer.com/10.1007/978-94-007-5860-5, doi:10.1007/978-94-007-5860-5.

[Int] International Organization for Standardization. UML ISO Sandard. URL: http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=32620.

[KG01] Ralf Kollmann and Martin Gogolla. *Capturing dynamic program behaviour with UML collaboration diagrams*. 2001. doi:10.1109/CSMR.2001.914969.

[KH14] Maximilian Koegel and Jonas Helming. EMF Tutorial, 2014. URL: http://eclipsesource.com/blogs/tutorials/emf-tutorial/.

[LGDW+11] Rose Louis M., Antonio Garcıa-Domınguez, James Williams R., Dimitrios Kolovos S., Richard F. Paige, and Fiona A.C. Polack. *Hello World with Epsilon*. 2011.

[LLLZ10] Tieqiang Li, Jianbin Liu, Manze Li, and Shuai Zhang. Research on information transformation based on XMI. In *Proceedings - 2010 3rd IEEE International Conference on Computer Science and Information Technology, ICCSIT 2010*, volume 6, pages 356–359, 2010. doi:10.1109/ICCSIT.2010.5563946.

[LZPH09] Gilles Lasnier, Bechir Zalila, Laurent Pautet, and Jérome Hugues. *Ocarina : An environment for AADL models analysis and automatic code generation for high integrity applications*. 2009. doi:10.1007/978-3-642-01924-1_17.

[Mai] Olaf Maibaum. *Software Evolution from TET-1 to Eu:CROPIS*.

[MLG] Olaf Maibaum, Daniel Lüdtke, and Andreas Gerndt. *Tasking Framework: Parallelization of Computations in Onboard Control Systems*.

[Mun14] M Munoz. *Space systems modeling using the Architecture Analysis amp; Design Language (AADL)*. IEEE International Symposium, 2014. doi:10.1109/ISSREW.2013.6688881.

[Niz02] Dionisio De Niz. Diagrams and Languages for Model-Based Software Engineering

of Embedded Systems : UML and AADL. page 10, 2002.

[Obja] Object Management Group. MetaObject Facility (MOF) Home Page. URL: http://www.omg.org/mof/.

[Objb] Object Management Group. Unified Modeling Language (UML). URL: http://www.uml.org/.

[Objc] Object Management Group. Webpräzens von SysML. URL: http://www.omgsysml.org/.

[Objd] Object Managment Group. Webpräsens von XMI. URL: http://www.omg.org/spec/XMI/.

[P&] P&P Software. The CORDET Project. URL: http://www.pnp-software.com/cordet/.

[Par10] Jesús Pardillo. *A systematic review on the definition of UML profiles*, volume 6394 LNCS. 2010. doi:10.1007/978-3-642-16145-2_28.

[PCD12] Maxime Perrotin, Eric Conquet, and Julien Delange. *TASTE : A Real-Time Software Engineering Tool-Chain Overview , Status , and Future*. 2012.

[Pro08] Herzberg Prof. Dr. Dominikus. Syntax und Semantik. *denkspuren*, November 2008. URL: http://denkspuren.blogspot.de/2008/11/syntax-und-semantik.html.

[PTDH11] Maxime Perrotin, Thanassis Tsiodras, Julien Delange, and Jérôme Hugues. *TASTE Documentation v1.1*. 2011.

[Ref11] Ivar Refsdal. *Comparison of GMF and Graphiti based on experiences from the development of the PREDIQT tool*. UNiversity of Oslo, 2011.

[RQZ12] Chris Rupp, Stephan Queins, and Barbara Zengler. *UML2 Glasklar*. Carl Hanser Verlag GmbH & Co. KG, 3. auflage edition, 2012.

[RR12] Zdenek Rybola and Karel Richta. *Using OCL in Model Validation According to Stereotypes*. 2012. URL: http://ceur-ws.org/Vol-837/paper3.pdf.

[SK95] Kenneth Slonneger and Barry L. Kurtz. *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach*. Addison-Wesley, 1995.

[TSB10] Skander Turki, Eric Senn, and Dominique Blouin. *Mapping the MARTE UML profile to AADL*. 2010.

[Tut] AADL Tutorial. AADL Processor and Memory Tutorial. Technical report. URL: http://aadltutorial.com/images/tutorial3/bus-without-connections.png.

[Ud] Uml-diagrams.org. UML Profile Diagrams. URL: http://www.uml-diagrams.org/profile-diagrams.html.

[W3C] W3C. Extensible Markup Language (XML). URL: http://www.w3.org/XML/.

[WTR] Jos Warmer, Karsten Thoms, and Joerg Reichert. *Spray User Guide Spray – A quick way of creating Graphiti User Guide*.

[ZJBA06] Rabih Zbib, Ashish Jain, Devasis Bassu, and Hiralal Agrawal. *Generating Domain Specific Graphical Modeling Editors from Meta Models*. 2006. doi:10.1109/COMPSAC.2006.48.